Introducción a Microservicios

El concepto de microservicios tiene sus orígenes en la arquitectura SOA (*Service Oriented Architecture*). SOA se basa en el antiguo principio de "divide y vencerás", y sostiene un modelo distribuido para el desarrollo de aplicaciones frente a soluciones clásicas más monolíticas.

Una arquitectura basada en SOA debe seguir una serie de principios para ser exitosa. Estos **principios** son:

- Cada servicio debe ofrecer un contrato para conectarse con él. Un caso muy común es un servicio que ofrece una API *REST*. Dicha API debe siempre mantener compatibilidad con versiones anteriores, o versionar sus *endpoints* cuando se producen incompatibilidades, pero es fundamental no romper el contrato con otros servicios.
- Cada servicio debe minimizar las dependencias con el resto. Para esto es fundamental acertar con el *scope* de un servicio. Una indicación de que el *scope* no es el adecuado es cuando se producen dependencias circulares entre los servicios.
- Cada servicio debe abstraer su implementación. Para el resto de servicios debe ser transparente si un servicio usa un *backend* u otro para la base de datos o si ha hecho una nueva *release*.
- Los servicios deben diseñarse para maximizar su reutilización dado que la reutilización de componentes es una de las ventajas de una arquitectura SOA.
- Cada servicio tiene que tener un ciclo de vida independiente, desde su diseño hasta su implantación en los entornos de ejecución.
- La localización física de donde corre un servicio debe ser transparente para los servicios que lo utilizan.
- En lo posible, los servicios deben evitar mantener estado.
- Es importante mantener la calidad de los servicios. Un servicio con continuas regresiones puede afectar a la calidad final percibida por el resto de servicios que hacen uso de él.

Teniendo en cuenta estos principios, el concepto de **microservicios** es un poco la manera que se ha puesto de moda para referirse a las arquitecturas SOA, pero incidiendo más aún en que la funcionalidad de dichos servicios debe ser la mínima posible. Una medida bastante extendida es que un microservicio es un componente que debería ser desarrollable en unas dos semanas. Las ventajas de una arquitectura basada en microservicios son las siguientes:

- Son componentes pequeños que agilizan los procesos de desarrollo de software y son fáciles de abordar por un equipo de desarrolladores.
- Son servicios independientes, si un microservicio falla no debería afectar a los demás.
- El despliegue de un microservicio a producción es más sencillo que el de una aplicación monolítica.
- Los microservicios son altamente reutilizables.
- Los microservicios son más fáciles de externalizar.

Dicho esto, una arquitectura de microservicios es sólo un modelo de desarrollo de software que mal aplicado puede traer enormes quebraderos de cabeza. Los microservicios adquieren más importancia cuando tenemos equipos de ingeniería muy grandes, que interesa dividir en subgrupos y cada uno de ellos se encargue de uno (o unos pocos) microservicios. Además, el proceso de migrar una arquitectura monolítica a una arquitectura basada en microservicios debe ser planeado con cautela. Se recomienda transferir un trozo de lógica a un sólo microservicio a la vez, ya que una arquitectura basada en microservicios puede implicar un cambio de las herramientas utilizadas para el despliegue, monitoreo y sistemas de *logging* de nuestras aplicaciones.

Microservicios antes y después de la aparición de los contenedores

Como hemos comentado, el despliegue de un microservicio es más sencillo que el de una aplicación monolítica debido a su sencillez. Sin embargo, los microservicios agilizan los procesos de desarrollo del

software, y pronto nos encontraremos con que al día podemos hacer varios despliegues de distintos microservicios. Por tanto, una arquitectura basada en microservicios es difícilmente concebible sin la automatización de los procesos de integración y despliegue continuo.

El boom experimentado por los contenedores ha facilitado muchísimo la automatización de estos procesos. **Docker** es la tecnología más extendida para la gestión de contenedores, y uno de las razones más importantes de su éxito (probablemente la que más) es la facilidad que ofrece para construir una imagen, distribuirla y ejecutarla en cualquier máquina independientemente de la infraestructura. Esto significa que podemos construir una imagen en nuestros entornos de integración continua, correr nuestras dockers contra ella, distribuirla en nuestro servidores de producción y por último, ejecutarla en un contenedor. Y todo esto ejecutando simplemente unos cuantos comandos de docker. En otras palabras, docker (y otras tecnologías de contenedores similares) presentan las siguientes ventajas:

- **Portabilidad**: un **contenedor** ejecuta lo que se denomina una **imagen**, que viene a ser una representación del sistema de ficheros (y otros metadatos) que el contenedor va a utilizar para su ejecución. Una vez que hemos generado una imagen, ya sea en nuestro ordenador o vía una herramienta externa, esta imagen podrá ser ejecutada por cualquier entorno siempre que dicho entorno soporte la ejecución de contenedores.
- **Inmutabilidad:** una aplicación la componen tanto el código fuente como las librerías del sistema operativo y del lenguaje de programación necesarias para la ejecución de dicho código. Estas dependencias dependen a su vez del sistema operativo donde nuestro código va a ser ejecutado, y por esto mismo ocurre muchas veces aquello de que *"no sé, en mi máquina funciona"* cuando luego han aparecido problemas en producción. Sin embargo, el proceso de instalación de dependencias de un contenedor se realiza de manera estática cuando se genera su imagen. Por tanto, una imagen siempre ejecutará con las misma versión del código fuente y sus dependencias, por lo que se dice que es inmutable. Esto unido a la portabilidad de los contenedores los conviertes en una herramienta fiable.
- **Ligereza:** los contenedores corriendo en la misma máquina comparten entre ellos el sistema operativo, pero cada contenedor es un proceso independiente con su propio sistema de ficheros y su propio espacio de procesos y usuarios. Por tanto, los contenedores son más ligeros que otros mecanismos de virtualización. Comparemos por ejemplo con Virtualbox, otra tecnología de virtualización. Virtualbox permite del orden de 4 ó 5 máquinas virtuales en un ordenador convencional, mientras que en el mismo ordenador podremos correr cientos de contenedores sin mayor problema, además de que su gestión es mucho más sencilla.

Como consecuencia, los contenedores se han mostrado como herramientas muy eficaces para optimizar la instalación de entornos locales para los desarrolladores, la simplificación de las tareas de integración continua y la automatización de los procesos de despliegue, todo ello aumentando la fiabilidad general del desarrollo del software, y por ello mismo han facilitado la implantación de arquitecturas basadas en microservicios.

Sin embargo, Docker y otras tecnologías de contenedores no solucionan todos los problemas que aparecen en una arquitectura basada en microservicios. Dichos problemas son:

- **Service Discovery**: debido a la cantidad de microservicios, estos deben ser fáciles de localizar por los servicios que quieran contactar con ellos. También debe ser sencillo (y seguro) conocer las credenciales para establecer dicha conexión.
- **Configuración de Red:** algunos de estos servicios sólo pueden ser contactados por un subconjunto de nuestros microservicios. Este problema suele solucionarse con la gestión de redes (y subredes) de tal manera que servicios en redes diferentes no pueden verse el uno al otro.
- **Persistencia:** algunos servicios inevitablemente tienen estado. Un contenedor puede ser fácilmente recreado en una nueva máquina si hay un fallo en el servidor donde se estaba ejecutando, pero lo mismo no aplica a los datos persistentes que utilizaba dicho contenedor.
- **Escalabilidad:** nuestras herramientas de orquestación y monitoreo debe escalar al número de contenedores desplegados en producción.

- **Configuración:** la herramienta que utilicemos debe ser fácilmente configurable con las subherramientas que sean más cómodas al equipo de operaciones.
- Logging y Monitoreo: debido a la gran cantidad de contenedores desplegados en producción debemos utilizar herramientas de monitoreo que nos faciliten identificar los contenedores donde se producen fallos del sistema.
- **Respuesta a fallo:** es importante monitorizar nuestros contenedores y desplegarlos en un nuevo servidor si hay una falla en la máquina donde estaban ejecutando.

Como hemos dicho, Docker no resuelve la mayoría de estas cuestiones, ya que su principal objetivo es facilitar la creación de imágenes, su distribución y su ejecución de una manera fiable. Es por esto que existen hoy en día multitud de herramientas de **gestión de clusters** que vienen a solucionar en lo posible estos problemas.

Soluciones para Gestión de Clusters *On Premise*

Conocemos como soluciones *on premise* aquellas que podemos instalar en nuestros propios servidores. Destacamos las siguientes:

Docker Swarm

Es la solución *open source* de Docker. Ha alcanzado la versión 1.0, pero aún le falta por solucionar algunos de los problemas más importantes como la gestión de fallos de infraestructura, mejorar el service discovery, las soluciones de red o las soluciones de monitoreo. Su principal ventaja es que tiene un *roadmap* muy ambicioso y que su API es compatible con la API del Docker Engine, por lo que todas las herramientas desarrolladas para Docker, que tiene el ecosistema más activo, funcionan directamente sobre Docker Swarm.

Nota: este comentario hace referencia a la versión de Docker 1.11. En las versiones de Docker 1.12 y posteriors, Docker Swarm pasa a formar parte del demonio de Docker, y ha mejorado mucho de los aspectos comentados aquí. Ver las clases de Docker 1.12 y Docker 1.13 para más información.

Kubernetes

Es la solución *open source* de Google. También ha alcanzado la versión 1.0 y su ecosistema es cada vez más activo. Es probablemente la solución más completa en la actualidad, con soluciones elegantes para service discovery, respuesta a fallo y la configuración de la red. Ofrece integraciones para utilizar volúmenes persistentes tanto en *Amazon Web Services* (AWS) como en *Google Compute Engine*, así como con soluciones de monitoreo (*cAdvisor*) y logging (*elasticsearch*). Su desarrollo es muy activo y también ha desarrollado soluciones para la gestión de secretos y la gestión de control de acceso. Su principal problema es que es una plataforma muy "opinionada", y es un error que Google ya ha cometido en otros proyectos como Google App Engine.

DCOS

Es la solución *open source* de Mesos. También ha alcanzado la versión 1.0 y presenta el respaldo de grandes marcas como Microsoft o HP. Su punto fuerte es que utiliza Mesos, una herramienta que da una visión de un único sistema operativo a un conjunto de máquinas, y que ha sido usada en producción en Twitter por varios años. Además, la interfaz de usuario está cuidada al detalle, aspecto donde Docker Swarm y Kubernetes no pueden estar muy orgullosos.

Soluciones para Gestión de Clusters *SaaS*:

Conocemos como soluciones *SaaS* aquellas que están disponibles como un servicio en internet. Destacamos las siguientes:

Docker Cloud

Es la solución *SaaS* de Docker e internamente utiliza un híbrido con Docker Swarm. La curva de aprendizaje es mínima si utilizas herramientas de Docker como *docker-compose*. Además de Docker Swarm, ofrece una solución para la configuración de redes basada en Weave que lo hace más potente, así como soluciones propias para service discovery basadas en DNS. Por último, ofrece soluciones de integración y despliegue continuo para automatizar los procesos desde que un nuevo *commit* está disponible en tu repositorio *git* hasta que los cambios llegan a producción. Está más centrada en desarrolladores ya que no tienen soluciones de control de acceso. Dispone de registro propio (*Docker Hub*).

GCE (Google Container Engine)

Es la solución *SaaS* de Google e internamente utiliza Kubernetes. Una vez desplegado el cluster de kubernetes, la interacción del usuario es directamente con el cluster usando la herramienta de línea de comandos de kubernetes. Por tanto, no ofrece grandes beneficios sobre kubernetes, más allá de facilitar su configuración con otros servicios del *Google Cloud Platform* como el logging o las métricas. Dispone de registro propio (*Google Container Registry*).

Azure Container Service

Es la solución *SaaS* de Microsoft e internamente utiliza DCSO. Aunque está en fase de desarrollo, tiene pinta de convertirse en una solución muy competitiva.

ECS (EC2 Container Service)

Es la solución *SaaS* de AWS e internamente utiliza una tecnología propietaria pero desarrollada sobre Docker. Se integra perfectamente con otros servicios de AWS como *Elastic Block Storage*, *Cloud Trail*, *Cloud Trail Logs*, *Identity and Access Management* o *Elastic Load Balancer*. Quizás sea la más difícil de utilizar como usuario de Docker, pero las integraciones con otros servicios de AWS le dan un plus frente a otras soluciones. Dispone de registro propio (*EC2 Container Registry*).

Conclusión

Hemos visto como las arquitecturas basadas en microservicios agilizan los procesos de desarrollo del software en equipos de ingeniería de mediano y gran tamaño. También hemos visto como Docker y los contenedores en general han dado un impulso definitivo a esta metodología de desarrollo del software, pero que aún quedan muchos problemas por resolver en un entorno de producción a gran escala. Es aquí donde aparecen las herramientas de gestión de clusters, que vienen a solucionar la mayoría (si no todos) de estos problemas. Es un campo en constante evolución y donde hay un rico y activo ecosistema de compañías colaborando en encontrar las mejores soluciones a estos problemas. En las próximas lecciones veremos algunas de las herramientas de mayor éxito en este contexto como son Kubernetes y Docker Swarm. DCOS es una herramienta muy interesante, pero obviamos su explicación por falta de tiempo.

o *Elastic Load Balancer*. Quizás sea la más difícil de utilizar como usuario de Docker, pero las integraciones con otros servicios de AWS le dan un plus frente a otras soluciones. Dispone de registro propio (*EC2 Container Registry*).

Hemos visto como las arquitecturas basadas en microservicios agilizan los procesos de desarrollo del software en equipos de ingeniería de mediano y gran tamaño. También hemos visto como Docker y los contenedores en general han dado un impulso definitivo a esta metodología de desarrollo del software, pero que aún quedan muchos problemas por resolver en un entorno de producción a gran escala. Es aquí donde aparecen las herramientas de gestión de clusters, que vienen a soluciones la mayoría (si no todos) de estos problemas. Es un campo en constante evolución y donde hay un rico y activo ecosistema de compañías colaborando en encontrar las mejores soluciones a estos problemas.

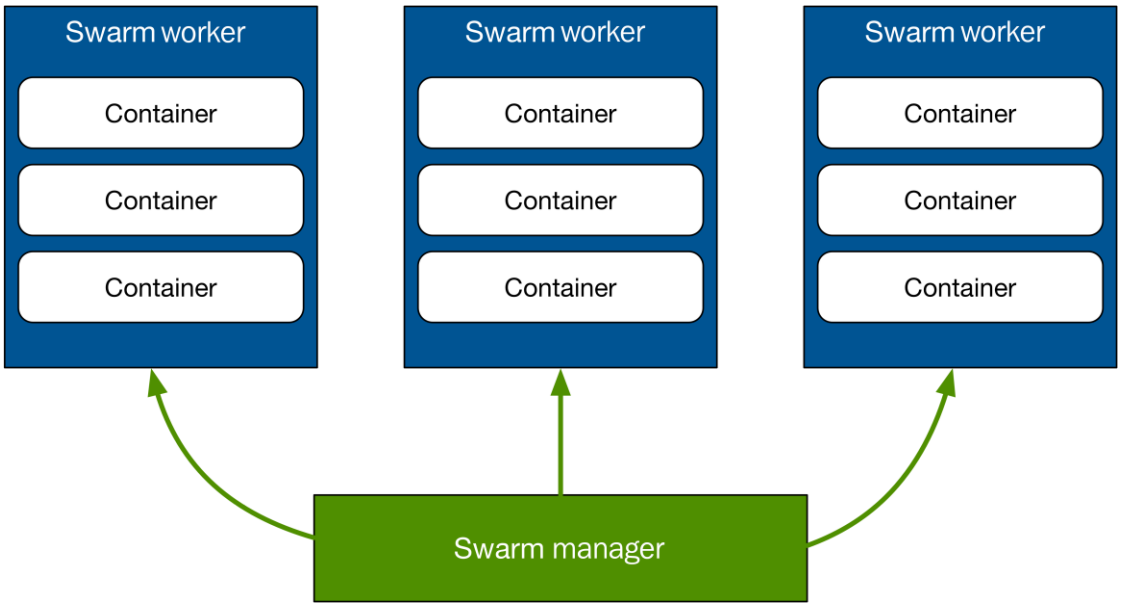
Docker Swarm

Anteriormente a la versión 1.12 de Docker, el uso de Docker Swarm era un componente separado, pero a partir de dicha versión forma parte de Docker Engine con el nombre de Swarm mode.

Qué es Swarm

Características principales:

- Se utiliza Docker Engine CLI para administrar swarm, por lo que no necesitamos ningún software adicional
- Diseño descentralizado: la direferenciación de los roles dentro del cluster se realiza en tiempo de ejecución, por lo que podemos utilizar la misma imagen para los diferentes roles
- Modelo declarativo: se utiliza el modo declarativo para describir los servicios.
- Escalado: por cada servicio se puede indicar el número de tareas o instancias a ejecutar. Gestión automatica tanto del aumento como la reducción de número de instancias por servicio
- DSC: el nodo de gestión monitoriza constantemente el estado del cluster y los nodos para corregir las posibles desviaciones del estado deseado declarado. Si, por ejemplo, se declara que un servicio tiene que ejecutar 4 instancias y se cae un nodo que ejecuta 2 de esas instancias, el manager se encarga de crear otras 2 instancias en el resto de los nodos disponibles.
- Red multi-host: swarm permite crear una red para los servicios que es común a todos los hosts (overlay network)
- Descubrimiento de servicios: para cada servicio que se ejecuta se asigna un nombre DNS único utilizando un servidor DNS embebido en swarm
- Balanceo de carga: se pueden exponer los puertos de los servicios a un balanceador de carga externo. Swarm permite definir como distribuir la carga de los servicios entre los nodos.
- Seguro por defecto: swarm utiliza TLS para la autenticación y encriptación de la comunicación entre los nodos, permitiendo la opción de utilizar certificados auto-firmados o de una CA personalizada.



Terminologina

Cuando trabajamos con Swarm es importante tener claro los siguientes términos:

- **Orquestación:** describe el flujo para desplegar aplicaciones relacionadas entre si.
- **Swarm:** es un cluster de Docker engine, también llamados nodos, donde se desplegaran servicios. El cliente de Docker incluye comandos para administrar nodos (añadir o eliminar), y desplegar y orquestar servicios a través de dichos nodos.
- **Nodo:** Un nodo es una instancian de Docker Engine que participara en Swarm. En entornos de producción habrá diferentes nodos separados en distintos servidores y en diferentes localizaciones.
- **Nodo administrador (Manger Node):** es el encargado de repartir las tareas a los nodos de trabajo.
- **Nodo de trabajo (Worker node):** recibe y ejecuta tareas repartidas por el nodo administrador.
- **Servicios:** un servicio es la definición de tareas a ejecutas en un modo de trabajo. Un servicio contiene la imagen a utilizar y los comandos a ejecutar dentro de los contenedores.
- **Tareas:** una tarea consiste en el comando a ejecutar dentro de un contenedor especifico.
- **Balanceador de Carga (Load Balancing):** el administrador de Swarm utiliza un balanceador de carga de conexiones entrantes para exponer los servicios deseados de forma pública. A través de un DNS interno automáticamente balanceara la carga a los contenedores de un servicio.

Conceptos clave

Es el módulo de Docker encargado de gestionar clusters y orquestar servicios. Está basado en swarmkit

Swarm está formado por múltiples hosts Docker que se ejecutan con uno de los dos roles:

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

Un host Docker puede ser manager, worker o los 2 roles.

Cuando creamos un servicio, definimos su estado óptimo:

- Número de replicas
- Red
- Recursos de almacenamiento
- Puertos expuestos
- …

Swarm se encarga de mantener ese estado óptimo gestionando la ejecución de los contenedores que mantienen esa configuración deseada.

A la ejecución de un contenedor que es parte de un servicio swarm se le denomina tareas (*task)*. Hay que diferenciar los contenedores que se ejecutan de forma individuarl (standalone) o los que se ejecutan como parte de un servicio swarm:

- En un host podemos ejecutar contenedores standalone y tasks
- Los cambios de configuración los gestiona swarm. Por ejemplo, ante un cambio de configuración de un servicio, es swarm quien actualiza la información, detiene las tareas que no están actualizadas y crea las nuevas con la nueva configuración. En un contenedor standalone estos pasos hay que realizarlos de forma manual.
- Los servicios swarm sólo se pueden ejecutar en un cluster swarm

Nodos (Nodes)

Un nodo es una instancia de Docker participando en swarm. Se pueden ejecutar uno o más nodos en un equipo físico, pero en un entorno típico se distribuyen los nodos por diferentes máquinas físicas.

Al desplegar una aplicación en swarm, indicamos la definición del servicio en un manager node. El manager node se encarga de que las unidades de trabajo (tasks) se ejecuten en los worker nodes.

El Manager node también se encarga de ejecutar las tareas destinadas a mantener el cluster y el estado deseado de los servicios. Pueden existir varios nodos manager en un cluster, y entre ellos se elegirá un lider encargado de las tareas de orquestación.

Los Worker nodes reciben y ejecutan las tareas indicadas por los manager nodes. Por defecto los manager nodes tambíen ejecutan tareas como worker nodes, pero se puede configurar que funcionen como manager nodes de forma exclusiva. Los worker nodes ejecutan un agente encargado de informar al manager del estado de las tareas que se le ha asignado.

Servicios y tareas (Services and tasks)

Un servicio (service) es la definición de las tareas que se ejecutan en un worker node. Es el objeto principal de la estructura que se define en un sistema swarm.

Cuando se crea un servicio se indica que imagen del contenedor se va a utilizar y que comandos se van a ejecutar en el contenedor.

En servicios replicados, se indca un número de tareas réplica que son necesarias en la ejecución y el manager node se encarga de que los worker nodes ejecuten ese número de instancias como estado deseado de ejecución.

Hay servicios globales de los cuales swarm se encarga de que se ejecute una instancia en cada uno de los nodos del cluster disponible. Pueden ser servicios globales agentes de monitorización, antivirus …

Una vez que una tarea se ha asignado a un nodo, no puede ser movida a otro nodo. O se ejecuta en ese nodo o muere.

Balanceo de carga (Load balancing)

Swarm manager utiliza balanceo de carga en el ingreso de tráfico para exponer los servicios que queremos que estén disponibles externamente. Swarm asigna automáticamente un puerto publicado al servicio (PublishedPort) o se puede configurar de forma manual. Si no se indica el puerto de forma manual, swarm manager asigna un puerto del rango 30000-32767

Componentes externos, como un balanceador de carga, pueden dar acceso al servicio a través del puerto publicado, accediendo a cualquiera de los nodos del cluster, indipendientemente de que ese nodo tenga tareas del servicio ejecutándose. Todos los nodos se encargan de enrutar las conexiones entrantes a una instancia de una tarea en ejecución.

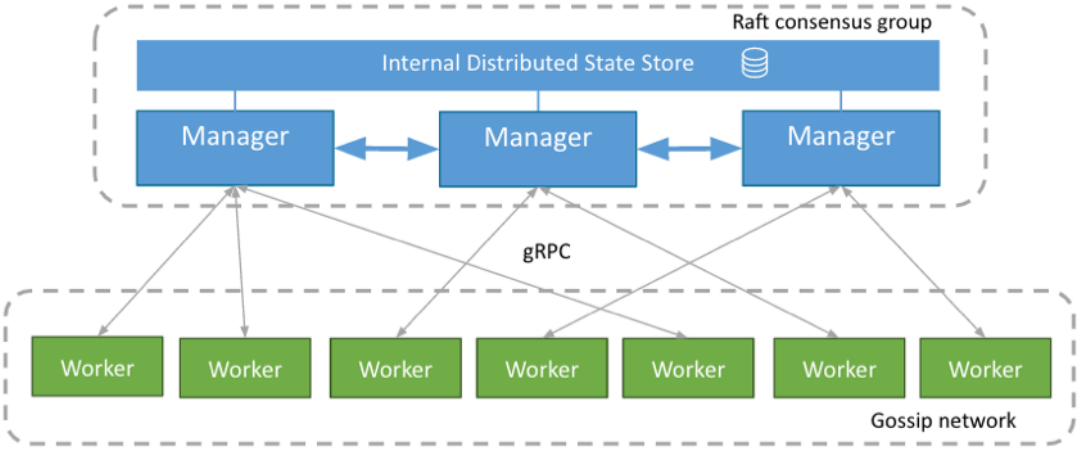
Swarm tiene un servidor DNS embebido internamente que asigna de forma automática nombres a cada servicio swarm. Swarm manager utiliza estos nombres para distribuir las peticiones de forma interna.

Arquitectura

Los requisitos de para desplegar nodos Swarm son los siguientes:

- Al menos una red para distribuir las comunicaciones, pero se recomienda tener 3 servidores: un nodo administrativo y dos nodos de trabajo.
- Doceker Engine 1.12 o posterior.
- Los siguientes puertos abiertos entre los servidores:
    - Puerto 2377/TCP para comunicaciones administrativas del cluster.
    - Puerto 7946/tcp y udp, para la comunicación entre nodos.
    - Puerto 4789/tcp y udp para trafico de red.

En la siguiente imagen se ve la arquitectura en un entorno de producción con 10 servidores, distribuidos como 3 nodos administrativos y 7 nodos de trabajo.



Servicios y tareas

Un **servicio** define las **tareas** que serán ejecutadas dentro del clúster.

Cuando creamos un servicio le indicamos a Swarm qué imagen y qué parametrización se utilizará para crear los **contenedores** que se ejecutarán después como tareas dentro del clúster.

Existen dos tipos de servicios, **replicados** y **globales**:

- En un servicio **replicado**, Swarm creará una tarea por cada réplica que le indiquemos para después distribuirlas en el clúster. Por ejemplo, si creamos un servicio con 4 réplicas, Swarm creará 4 tareas.
- En un servicio **global**, Swarm ejecutará una tarea en cada uno de los nodos del clúster.

Como hemos dicho antes, las **tareas** son la **unidad de trabajo** dentro de Swarm. Realmente son la suma de un contenedor más el comando que ejecutaremos dentro de ese contenedor.

Los Manager **asignan** tareas a los nodos Worker de acuerdo al número de réplicas definidas por el servicio. Una vez que la tarea es asignada a un nodo ya no se puede mover a otro, tan sólo puede ejecutarse o morir.

Ante la **caída** de una tarea, Swarm es capaz de crear otra similar en ese u otro nodo para cumplir con el número de réplicas definido.

Balanceo

Swarm tiene un sistema de **balanceo** interno para exponer servicios hacia el **exterior** del clúster.

Un Manger es capaz de **publicar** automáticamente un puerto generado al azar en el rango *30000-32767* para cada servicio, o bien, nosotros podemos publicar uno específico.

Cualquier sistema externo al clúster podrá acceder al servicio en este puerto publicado a través de **cualquier** nodo del clúster, independientemente de que ese nodo esté ejecutando una tarea del servicio o no.

Todos los nodos del clúster **enrutarán** a una tarea que esté ejecutando el servicio solicitado.

Además, Swarm cuenta con un **DNS** interno que asigna automáticamente una entrada a cada uno de los servicios desplegados en el clúster.

En nuestro laboratorio utilizaremos los siguientes servidores:

Nombre	Descripción	Dirección IP
Docker	Nodo administrador	192.168.1.150
Docker2	Nodo de trabajo	192.168.1.152
Docker3	Nodo de trabajo	192.168.1.153

Todos los servidores se están resolviendo entre ellos a través del fichero **/etc/hosts**

Vamos ha habilitar los siguientes puertos en el cortafuegos **no seria necesarios para nuestro lab**:

- **TCP port 2377** for cluster management communications
- **TCP** and **UDP port 7946** for communication among nodes
- **UDP port 4789** for overlay network traffic

```
#firewall-cmd --add-port=2377/tcp --permanent
#firewall-cmd --add-port=7946/tcp --permanent
#firewall-cmd --add-port=7946/udp --permanent
#firewall-cmd --add-port=4789/udp --permanent
```

Creación del clúster

Para crear un clúster con **Swarm Mode** tenemos que partir de un nodo destinado a ser **Manager**. Este nodo debe tener Docker 1.12 o superior ya instalado.

$ docker swarm init --advertise-addr <ip-nodo administrador>

Suponiendo que la IP del nodo es *192.1681.150*, ejecutamos el siguiente comando:

[root@docker ~]# docker swarm init --advertise-addr 192.168.1.150

Error response from daemon: --live-restore daemon configuration is incompatible with swarm mode

Cuando lanzamos este comando nos encontramos con este error, tendremos que realizar los siguientes **cambios en todos los nodos** que van a conformar el cluster de Swarm:

#vi /etc/docker/daemon.json

{
"live-restore": **false**
}

 # systemctl restart docker

https://forums.docker.com/t/error-response-from-daemon-live-restore-daemon-configuration-is-incompatible-with-swarm-mode/28428/3

[root@docker ~]# **docker swarm init --advertise-addr 192.168.1.150**

Swarm initialized: current node (6nh15zseil4zq1an1oww5fjo6) is now a manager.

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-1jm078wfk8e2fq0sikf4atznyqdkpkei7ddynugekej2kbs6u6-3hiesreb2tf8ctwv28c0r61pd \
192.168.1.150:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

Utilizando la acción info podemos obtener el estado actual de Swarm:

[root@docker ~]# docker info

Podemos listar los nodos de Swarm utilizando docker node ls:

[root@docker ~]# docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

6nh15zseil4zq1an1oww5fjo6 * docker Ready Active Leader

Añadir los nodos de trabajo:

En la salida de la **acción init** nos mostraba el comando que debemos ejecutar en los nodos de trabajo para añadirnos a Swarm. En el caso de no tener la salida disponible es posible volver a obtenerla con el siguiente comando desde el nodo administrador:

[root@docker ~]# docker swarm join-token worker

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-1jm078wfk8e2fq0sikf4atznyqdkpkei7ddynugekej2kbs6u6-3hiesreb2tf8ctwv28c0r61pd \
192.168.1.150:2377
```

Debemos ejecutar dicho comando en todos los nodos de trabajos que queremos añadir al cluster:

**[root@docker2 ~]# docker swarm join **

**> --token SWMTKN-1-1jm078wfk8e2fq0sikf4atznyqdkpkei7ddynugekej2kbs6u6-3hiesreb2tf8ctwv28c0r61pd **

> 192.168.1.150:2377

This node joined a swarm as a worker.

**[root@docker3 ~]# docker swarm join **

**> --token SWMTKN-1-1jm078wfk8e2fq0sikf4atznyqdkpkei7ddynugekej2kbs6u6-3hiesreb2tf8ctwv28c0r61pd **

> 192.168.1.150:2377

This node joined a swarm as a worker.

Una vez configurados los nodos de trabajo, podemos listar su estado desde el nodo administrador con el comando:

[root@docker ~]# docker node ls

```
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
4tz8xgw7rzftiv9m6au5a5jqi    docker2   Ready   Active
6nh15zseil4zq1an1oww5fjo6 *  docker    Ready   Active        Leader
atervszajg72xa3emii49anng    docker3   Ready   Active
```

En la columna **MANAGER STATUS** observamos que nodos son administrador, podremos tener varios y uno de ellos será el líder (Leader).

Obtener información de los nodos dentro de swarm:

```
# docker node inspect docker –pretty
# docker node inspect docker2 --pretty
```

Desplegando un Servicio

Una vez configurado nuestro nodo administrador y nuestros nodos de trabajo, todo estará listo para desplegar nuestros servicios. Para ello utilizaremos de **acción service** créate que tiene la siguiente sintaxis:

$ docker service créate [opciones] imagen comando

Donde las opciones básicas son las siguientes:

- **--name nombre:** el nombre para el servicio.
- **--replicas numero:** el numero de nodos a desplegar el servicio.
- **p/--publish puerto:** el puerto a publicar para ser accesible.

En este punto lanzamos un servicio llamado www, (replicas 1), del contenedor nginxdemos/hello, :

[root@docker ~]# docker service create --name www -p80:80 --replicas 1 nginxdemos/hello

```
cwms9fqnuvthwxtljk4f8bu5z
```

Para listar los servicios y el estado de ellos, ejecutamos el siguiente comando:

[root@docker ~]# docker service ls

```
ID          NAME  REPLICAS  IMAGE           COMMAND
cwms9fqnuvth  www   1/1       nginxdemos/hello
```

Para obtener información detallada sobre el servicio, al igual que con otros objetos de Docker, utilizaremos la **subaccion inspect** (la **opción –pretty** muestra la salida sin ser en formato JSON).

[root@docker ~]# docker service inspect --pretty www

ID: cwms9fqnuvthwxtljk4f8bu5z

Name: www

Mode: Replicated

Replicas: 1

Placement:

UpdateConfig:

Parallelism: 1

On failure: pause

ContainerSpec:

Image: nginxdemos/hello

Resources:

Ports:

Protocol = tcp

TargetPort = 80

PublishedPort = 80

Al ejecutar este servicio, le especificamos que se ejecutara solo en un nodo (--replicas 1); para ver en cual se está ejecutando actualmente utilizamos la **subacion ps**

[root@docker ~]# docker service ps www

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR

788kuqm0k478vtnenunty1egu www.1 nginxdemos/hello docker Running Running 5 minutes ago.

Si accedemos al nodo de trabajo donde se esta ejecutando dicho servicio, en este caso docker, podemos listar los contenedores ejecutándose con el comando:

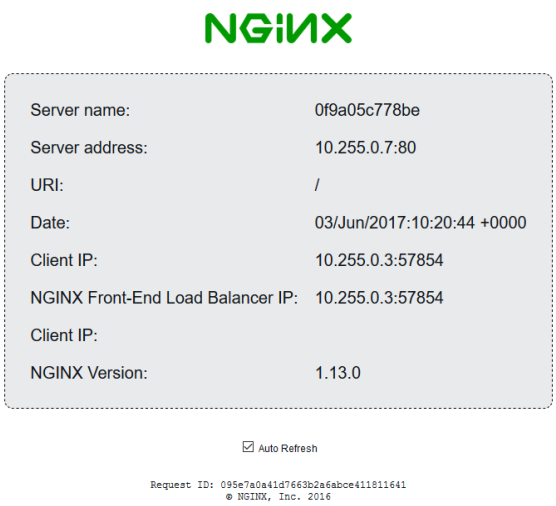
[root@docker ~]# docker ps -all

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

0f9a05c778be nginxdemos/hello:latest "nginx -g 'daemon off" 7 minutes ago Up 7 minutes 80/tcp www.1.788kuqm0k478vtnenunty1egu

Si accedemos a la ip del nodo manager al puerto 80 que es donde hemos casado el puerto del contenedor, vermos como al minuto balancea entre los contenedores:

http://192.168.1.150/



Ahora desplegamos un servicio pero tiene que correr en un nodo que sea worker:

```
#docker service create --name cluster1 --constraint "node.role == worker" -p:81:80/tcp russmckendrick/cluster
```

Comprobamos que el contenedor corre en un servidor worker:

```
# docker service ps cluster1
```

```
http://192.168.1.152:81/
```

Escalar un servicio

En la creación del servicio anterior (www), hemos especificado que solo se utilice una replica de la aplicación a desplegar a partir de una imagen. Es posible después de la creación, escalar para que se desplieguen mas contenedores distribuido a través de Swarm. Para ello utilizaremos el comando:

$ docker service scale servicio=replicas

En el siguiente ejemplo desplegaremos 4 replicas al servicio creado previamente:

[root@docker ~]# docker service scale www=4

www scaled to 4

[root@docker ~]# docker service ps www

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR

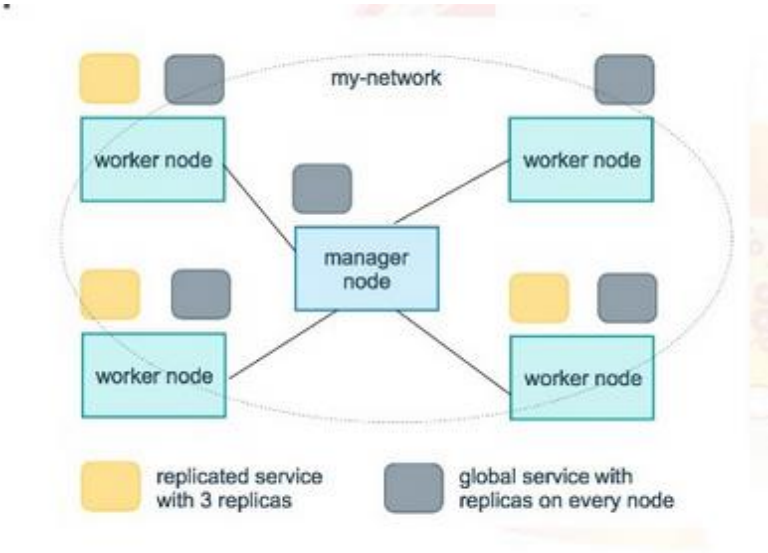
788kuqm0k478vtnenunty1egu www.1 nginxdemos/hello docker Running Running 12 minutes ago

5z358c4amc7bk7wcmafxeqgac www.2 nginxdemos/hello docker3 Running Running 3 seconds ago

26l3k2gv37517xs4ayv9tc4cu www.3 nginxdemos/hello docker2 Running Running 2 seconds ago

6osfh1w5voxs7ady2mvndt1mv www.4 nginxdemos/hello docker2 Running Running 3 seconds ago

Observarnos que los contenedores se han desplegado a través de todos los nodos que forman el cluster de Swarm.



Obtener los contenedores que corren en los nodos del cluster desde el Manager:

```
[root@docker ~]# docker node ps docker2
[root@docker ~]# docker node ps docker
```

Eliminar un servicio

Una vez que un servicio ya no sea necesario, podemos eliminarlo como se muestra a continuación:

[root@docker ~]# docker service rm www

www

[root@docker ~]# docker service ps www

Error: No such service: www

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

[root@docker2 ~]# docker ps -a

CONTAINER ID IMAGE

[root@docker3 ~]# docker ps -a

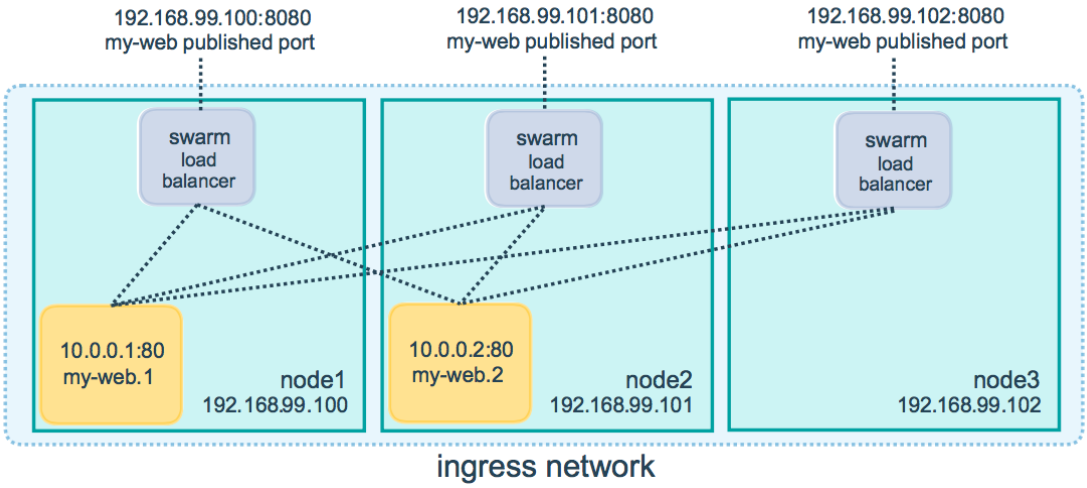
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Pasado un tiempo después de la eliminación del servicio, los contenedores que se ejecutaban en los nodos serán eliminados.

Publicar Puertos

Para permitir el acceso a los servicios swarm utiliza una malla de red **(routing mesh)** que permite acceder a los servicios publicados utilizando la dirección de cada uno de los nodos, aunque ese nodo no tenga ninguna tarea del servicio ejecutándose, en este caso, swarm se encargará de redirigir la petición a una de las tareas que se ejecuta en otro nodo.

Al crear un servicio podemos especificar el puerto que queremos publicar y será accesible desde todos los nodos. En la siguiente imagen describe la arquitectura al ejecutar un servidor web con 2 replicas y publicados el puerto 80 a través del puerto 8080 de los nodos:



Podemos observar que el puerto 8080 es accesible en todos los nodos y ejercerá como balanceador de carga

[root@docker ~]# docker service create --name servicio1 --replicas 3 --publish 8080:80 dockercloud/hello-world

chcy2jqu8hpjquml4cyj93f91

[root@docker ~]# docker service ps servicio1

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
26bvahqhsjr1jxqppsxgqd3do	servicio1.1	dockercloud/hello-world	docker	Running	Running about a minute ago	
5bexwybzkdb8xwboicdz9bte9	servicio1.2	dockercloud/hello-world	docker2	Running	Running about a minute ago	
3cg1l3fi4lt8mw9g6wvcvtrei	servicio1.3	dockercloud/hello-world	docker3	Running	Running about a minute ago	

Publicar un puerto para un servicio

Para permitir el acceso a un servicio tenemos que utilizar el parámetro ***–publish***.

Al crear el servicio, utilizamos:

```
#docker service create \
  --name <SERVICE-NAME> \
  --publish published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <IMAGE>
```

- *–publish*: es el parámetro para publicar el puerto
- *published=<PUBLISHED-PORT>*: es el puerto publicado y que utilizaremos para acceder al servicio. Si no se indica se utiliza un puerto aleatorio.
- *target=<CONTAINER-PORT>*: es el puerto utilizado por el contenedor

Para modificar un servicio ya creado, utilizamos:

```
#docker service update \
  --publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <SERVICE>
```

Podemos indicar si vamos a utilizar TCP o UDP para cada uno de los puertos:

- TCP

```
#docker service create --name dns-cache \
  --publish published=53,target=53 \
  dns-cache
```

- TCP y UDP

```
#docker service create --name dns-cache \
  --publish published=53,target=53 \
  --publish published=53,target=53,protocol=udp \
  dns-cache
```

- UDP

```
#docker service create --name dns-cache \
  --publish published=53,target=53,protocol=udp \
  dns-cache
```

Ejemplo de publicación

Siguiendo con el ejemplo que estabamos utilizando **dockercloud/hello-world** , vamos a recrear el servicio en el cluster swarm, pero esta vez vamos a crear 2 instancias

```
#docker service create --replicas 2 --name web3 --update-delay 20s --
publish published=8080,target=80 dockercloud/hello-world
```

De esta forma, cuando accedemos al puerto 8080 de cualquiera de los hosts, Docker redirecciona la petición al puerto 80 de alguno de los contenedores de las tareas que se ejecutan en los nodos.

Para comprobar los accesos, vamos a acceder a las ips de cada uno de los hosts y comprobar que accediendo a cualquiera de ellos, accedemos al servicio.

Enrutamiento en modo host

Si no queremos utilizar el modo “malla” (mesh) podemos utilizar el modo “host” que nos permite:

- Si accedemos a la IP de un nodo, accedemos siempre a una de las tareas que se ejecutan en ese nodo
- Si un nodo no tiene tareas del servicio ejecutando, el servicio no escucha en ese puerto en el nodo
- Si utilizamos más tareas que nodos, no podemos especificar un puerto específico de escucha.

Para cambiar el modo, utilizamos el parámetro *mode*, por ejemplo:

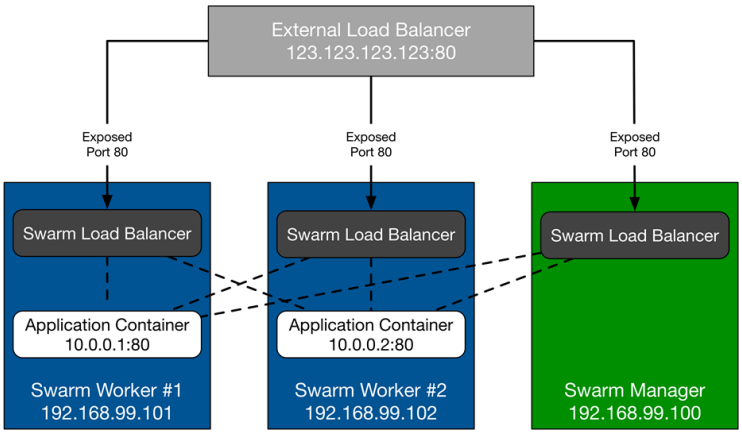
```
#docker service create --replicas 1 --name web4 --update-delay 20s --
publish published=8080,target=80,mode=host dockercloud/hello-world
```

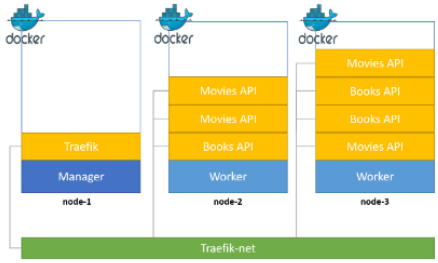
Comprobamos en que nodo esta corriendo en contenedor y verificamos el resultado:

```
# docker service ps web4
```

Utilizar un balanceador externo

La forma más común de acceder a un servicio del que tenemos varias tareas replicadas por varios nodos es utilizar un balanceador externo. Por ejemplo, podemos publicar con HAProxy,Traefick, en una dirección IP externa el virtual server que utiliza los servicios de los 3 nodos.





Eliminar nodos

Para eliminar nodos utilizaremos la **subaccion leave**, ejecutando el comando en el servidor donde queremos realizar dicha acción. Se puede ejecutar dentro de un nodo de trabajo o en un nodo administrador

[root@docker3 ~]# docker swarm leave

[root@docker2 ~]# docker swarm leave

[root@docker~]# docker node ls

Si ejecutamos la acción en un nodo administrador siendo el único, toda la configuración del cluster se borrara.

Para eliminar un node de la configuración, utilizaremos la siguiente configuración:

[root@docker~]# docker node rm docker3

[root@docker~]# docker node ls

Otros Comandos;

Si lanzamos este comando en el nodo manager, no lanzara contenedores en el managesr, solo los lanzara en los workers.

[root@docker ~]# **docker node update --availability drain docker**

Promocionar un nodo de trabajo a nodo administrador:

[root@docker2 ~]# **docker node promote docker2**

[root@docker2 ~]# **docker node ls**

Degradar un nodo administrador a nodo de trabajo:

[root@docker ~]# **docker node demote docker**

[root@docker ~]# **docker node ls**

Lanzar un servicio en modo global y limitando la memoria:

- El modo servicio tiene dos opciones:
  - Replicated (por defecto)
  - Global

El modo global creara un contenedor en cada nodo, no es compatible con –replicas.

[root@docker ~]# **docker service créate –mode global –name servicio4 –limit-memory 128MB –publish 8083:80 httpd**

GESTIÓN DE SWARM

```
root@docker:~$ docker swarm

Usage: docker swarm COMMAND

Manage Swarm

Options:
      --help   Muestra su uso

Commands:
  init        Inicia un Swarm
  join        Unir nodo a un Swarm y/o Manager
  join-token  Administra los tokens de unión
  leave       Sale del Swarm
  unlock      Desbloquea Swarm
  unlock-key  Administra la clave de desbloqueo
  update      Actualiza el Swarm

Run 'docker swarm COMMAND --help' for more information on a command.
```

Iniciar un Swarm

El comando `docker swarm init` permitirá iniciar un Docker Swarm. Sin embargo, en la mayoría de las ocasiones será necesario especificar un dirección IP o tarjeta de red para que el clúster pueda iniciarse debido a que se disponga de varias interfaces de red.

```
root@docker:~$ docker swarm init
Error response from daemon: could not choose an IP address to advertise since this system has
multiple addresses on different interfaces (10.0.2.15 on eth0 and 192.168.1.150 on eth1) -
specify one with --advertise-addr
```

Así pues, podemos hacerlo mediante la especificación de la tarjeta de red con `docker swarm init --advertise-addr eth1` o la dirección IP del nodo a través del parámetro *--advertise-addr*

```
root@docker:~$ docker swarm init --advertise-addr 192.168.1.150
Swarm initialized: current node (tpznikyxkv27wyu7bzvblfoyd) is now a manager.


To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r91c-
6arxrguisp3wa7zmubzms8bqn \
    192.168.1.150:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

Iniciar un Swarm con autolock

Autolock es una opción que permite cifrar Swarm a través de un token. Esto se realiza pasándole el parámetro al iniciar el nodo de Swarm, no puede hacerse a posteriori.

Aquí ejecutamos un nuevo Swarm con autolock y utilizando el nombre de la interfaz de red y no la dirección IP como lo hacemos en el punto *2.9.1*.

```
root@docker:~$ docker swarm init --autolock --advertise-addr eth1
Swarm initialized: current node (m226dmt32l13h9i5nzq5ai62o) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-1dabx0g90mdmgak3kyrat74ivmf5ileypbb51tl1msoo5g8ypx-
476s0g29uxdzmvf724o55djm7 \
    192.168.1.150:2377
```

```
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.

To unlock a swarm manager after it restarts, run the `docker swarm unlock`
command and provide the following key:

    SWMKEY-1-0A2xk5lptIcO3N+xKTYSfRNJl/aM9A7sK4pU3z4Ex7Y

Please remember to store this key in a password manager, since without it you
will not be able to restart the manager.
```

Para que los cambios surtan efecto es necesario que se reinicie el servicio de Docker Swarm. En nuestro caso lo hemos realizado reiniciando el nodo de VirtualBox completo.

```
root@docker:~$ exit
exit status 1
root@docker:~$ docker-machine restart docker
Restarting "docker"...
(docker) Check network to re-create if needed...
(docker) Waiting for an IP...
Waiting for SSH to be available...
Detecting the provisioner...
Restarted machines may have new IP addresses. You may need to re-run the `docker-machine env`
command.
```

Desbloquear un Swarm

En el punto anterior hemos iniciado un Swarm con el parámetro *--autolock*, esto hace que su contenido esté cifrado y no sea accesible tras un reinicio sin desbloquearlo.

```
root@docker:~$ docker swarm join-token manager
Error response from daemon: Swarm is encrypted and needs to be unlocked before it can be
used. Please use "docker swarm unlock" to unlock it.
root@docker:~$ docker swarm unlock
Please enter unlock key:
root@docker:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-1dabx0g90mdmgak3kyrat74ivmf5ileypbb51tl1msoo5g8ypx-
8ny7qh2wcul4vtr1qqqz1i5y7 \
    192.168.1.150:2377
```

Lógicamente, si no conocemos el token de desbloqueo no podremos administrar el Swarm. Este token podemos consultarlo a través de `docker swarm unlock-key -q` y no puede ser cambiado.

Sí podremos hacer que el nodo abandone el Swarm con `docker swarm leave --force`

Administrar tokens de unión

Los tokens son los que permitirán a otros nodos unirse al Docker Swarm. Estos son proporcionados por un nodo manager y deben ejecutarse dentro del nodo que queremos añadir al Docker Swarm.

Aquí vemos un ejemplo de qué ocurriría si solicitásemos los tokens a un nodo que no es Manager o a un nodo que no forma parte del Swarm.

```
root@docker2:~$ docker swarm join-token worker
Error response from daemon: This node is not a swarm manager. Worker nodes can't be used to
view or modify cluster state. Please run this command on a manager node or promote the
current node to a manager.

root@docker2:~$ docker swarm leave
Node left the swarm.

root@docker2:~$ docker swarm join-token worker
Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or
"docker swarm join" to connect this node to swarm and try again.
```

Con `docker swarm join-token manager` obtendremos el comando a ejecutar para añadir otro nodo manager. Mientras que `docker swarm join-token worker` nos daría el resultado a ejecutar en otro nodo que quisiéramos añadir como worker.

```
root@docker:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r9lc-
9cbal67yfh46urqrcw20r65kd \
    192.168.1.150:2377

root@docker:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r9lc-
6arxrguisp3wa7zmubzms8bqn \
    192.168.1.150:2377
```

Podemos obtener únicamente el token con el parámetro *-q*, esto permitiría tratar el token con procesos de automatización.

```
root@docker:~$ docker swarm join-token worker -q
SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r9lc-6arxrguisp3wa7zmubzms8bqn
```

Unir un nodo a un Swarm

Para unir un nodo a un Swarm necesitamos los tokens que proporciona el nodo Manager.

En este caso añadimos el nodo *docker2* como Manager del Swarm.

```
root@docker2:~$ docker swarm join \
>     --token SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r9lc-
9cbal67yfh46urqrcw20r65kd \
>     192.168.1.150:2377

This node joined a swarm as a manager.
```

Ahora, también será posible consultar los tokens en este nodo, ya que cumple la función de Manager.

```
root@docker2:~$ docker swarm join-token worker -q
SWMTKN-1-278pxc9p8ybm68xo0kuchigijtrugn1q80w5js5ae4i0n0r9lc-6arxrguisp3wa7zmubzms8bqn
```

Salir de un Swarm

Podemos hacer que un nodo abandone el Swarm, dejando de prestar servicio en el clúster de Docker. Si se trata de un nodo Manager deberemos forzar la acción, ya que esto haría que el resto del clúster no funcionase.

```
root@docker2:~$ docker swarm leave
Error response from daemon: You are attempting to leave the swarm on a node that is
participating as a manager. Removing the last manager erases all current state of the swarm.
Use `--force` to ignore this message.

root@docker2:~$ docker swarm leave --force
Node left the swarm.
```

Esto puede ser una opción para aprovechar recursos de máquinas en picos de demanda y que luego seguirán ejecutando su función normal, sin formar parte de un clúster de Docker.

GESTIÓN DE NODOS

```
root@docker:~$ docker node

Usage: docker node COMMAND

Manage Swarm nodes

Options:
      --help   Muestra su uso

Commands:
  demote      Quita uno o más nodos como Manager en el Swarm
  inspect     Muestra información detallada de uno o más nodos
  ls          Lista los nodos en el Swarm
  promote     Promueve uno o más nodos como Manager en el Swarm
  ps          Lista las tareas corriendo en uno o más nodos, por defecto en el nodo actual
  rm          Elimina uno o más nodos del Swarm
  update      Actualiza un nodo

Run 'docker node COMMAND --help' for more information on a command.
```

Promover nodo a Manager

Desde un nodo Manager es posible gestionar los roles del resto de ndos de Swarm, de manera que podemos promover un nodo Worker a Manager o quitarle ese rol.

```
root@docker:~$ docker node   ls
ID                            HOSTNAME            STATUS              AVAILABILITY
MANAGER STATUS
m226dmt32l13h9i5nzq5ai62o *   docker              Ready               Active
Leader




zc0suyc5ompi8agkt5tpg4bkd     docker2            Ready               Active

root@docker:~$ docker node promote docker2
Node docker2 promoted to a manager in the swarm.


root@docker:~$ docker node   ls
ID                            HOSTNAME            STATUS              AVAILABILITY


MANAGER STATUS
m226dmt32l13h9i5nzq5ai62o *   docker              Ready               Active
Leader
zc0suyc5ompi8agkt5tpg4bkd     docker2             Ready               Active
Reachable
```

El nodo *docker2* pasará a ser Manager, pero el nodo *docker* seguirá cumpliendo las funciones de *Leader*. Aún así, *docker2* puede gestionar los roles de la misma manera que *docker*, por lo que podría quitar el rol de Manager a cualquier otro nodo.

También podemos hacerlo con el comando `docker node update --role manager docker2`

Degradar nodo a Worker

Siguiendo con el ejemplo expuesto anteriormente, ahora quitaremos el rol de Manager al nodo *docker* (que concedió rol de Manager a *docker2*).

De esta manera comprobamos cómo el nodo *docker2* puede gestionar el rol del nodo Leader (*docker*) y como pasa a asumir el estado de Leader.

```
root@docker2:~$ docker node ls
ID                           HOSTNAME            STATUS              AVAILABILITY
MANAGER STATUS
m226dmt32l13h9i5nzq5ai62o    docker              Ready               Active
Leader
zc0suyc5ompi8agkt5tpg4bkd *  docker2             Ready               Active
Reachable

root@docker2:~$ docker node demote docker
Manager docker demoted in the swarm.

root@docker2:~$ docker node ls
ID                           HOSTNAME            STATUS              AVAILABILITY
MANAGER STATUS
m226dmt32l13h9i5nzq5ai62o    prueba              Ready               Active
zc0suyc5ompi8agkt5tpg4bkd *  prueba02            Ready               Active
```

También puede hacerse con el propio nodo.

```
root@docker2:~$ docker node promote docker Node
docker promoted to a manager in the swarm.
root@docker2:~$ docker node demote docker2
Manager docker2 demoted in the swarm.
```

O con el comando `docker node update --role manager docker2`

Listar los contenedores de un nodo

Para listar los contenedores que se están ejecutando en un nodo utilizamos el comando `docker node ps` desde un nodo Manager.

Si lo hacemos sin parámetros se mostrará los contenedores que se ejecutan en el nodo actual, que mostraría un resultado similar a ejecutar `docker ps`

```
root@docker:~$ docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE           ERROR               PORTS
tg6cpua4pya0        registry.1          registry:latest     docker              Running
Running 22 seconds ago
5u8bzvwf2yus        registry.3          registry:latest     docker              Running
Running 22 seconds ago

root@docker:~$ docker node ps docker2
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE           ERROR               PORTS
my5rr4qae5yo        registry.2          registry:latest     docker2             Running
Running 57 seconds ago
```

```
feyjwb8zacj3        registry.4          registry:latest     docker2             Running
Running 57 seconds ago
483yupyxd5hb        registry.5          registry:latest     docker2             Running
Running 57 seconds ago
```

Un nodo Worker no puede ejecutar `docker node ps` ni en su propio nodo. Para consultar los contenedores que están corriendo en un nodo Worker debemos hacerlo desde un Manager.

Volvemos a dejar al nodo docker como manager.

[root@nodo2 ~]# **docker node promote docker**

[root@docker ~]# **docker node ls**

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
of018zaa42u3ck7u073vaaujk *	docker	Ready	Active	Reachable	18.06.0-ce
bl5dq5s57gbjrcrz8ug1quk42	nodo2.curso.local	Ready	Active	Leader	18.06.0-ce
welytsefg5ed2qqqmko6zfp14	nodo3.curso.local	Ready	Active		18.06.0-ce

Cambiar nodo a solo Manager y no Manager+Worker

Por defecto, un nodo Manager también actúa como nodo Worker debido a que su disponibilidad es *Active*. Así pues, todos los nodos de un Swarm ejecutarán en él tareas desplegadas por un servicio y, por lo tanto, ejecutará en él contenedores.

Este estado se indica en el valor *Availability*.

Para que un nodo Manager no ejecute ninguna tarea debemos cambiar su disponibilidad a *Drain*. Lo hacemos con la ejecución `docker node update --availability drain docker` en nuestro nodo Manager.

```
root@docker:~$   docker node update --availability drain docker
docker
root@docker:~$   docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE            ERROR               PORTS
tg6cpua4pya0        registry.1          registry:latest     docker              Shutdown
Shutdown 31 seconds ago
5u8bzvwf2yus        registry.3          registry:latest     docker              Shutdown
Shutdown 31 seconds ago

root@docker:~$ docker      docker2
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE            ERROR               PORTS
wbp9zkr01dms        registry.1          registry:latest     docker2             Running
Running 42 seconds ago
my5rr4qae5yo        registry.2          registry:latest     docker2             Running
Running 11 minutes ago
n3ybk7qbljhj        registry.3          registry:latest     docker2             Running
Running 42 seconds ago
feyjwb8zacj3        registry.4          registry:latest     docker2             Running
Running 11 minutes ago
483yupyxd5hb        registry.5          registry:latest     docker2             Running
Running 11 minutes Ago
root@docker:~$ docker   ls
ID                           HOSTNAME            STATUS              AVAILABILITY
MANAGER STATUS
m226dmt32l13h9i5nzq5ai62o *   docker              Ready               Drain
Leader
zc0suyc5ompi8agkt5tpg4bkd     docker2             Ready               Active
```

El valor de la columna "Availability" pasará de *Active* a *Drain* y los contenedores que se estaban ejecutando en *docker* se ejecutarán ahora en *docker2*, "vaciando" de contenedores el nodo Manager.

También podemos evitar que ciertos servicios se ejecuten en un nodo concreto, en este caso en el nodo manager, a través del uso del parámetro *constraint* de *docker service*.

Cambiar disponibilidad de un nodo

En el anterior punto se editaba la disponibilidad (valor *Availability*) para que el nodo Manager no ejecutara ninguna tarea, estableciendo el valor a *Drain*.

Esta columna puede tener dos valores distintos más: *Active* y *Pause*.

 # docker node ls
 # docker node update --availability

Active es el valor por defecto para todos los nodos, hará que ese nodo reciba tareas del Swarm para ejecutar en él. Mientras que *Pause* hará que el nodo deje de recibir nuevas tareas, pero manteniendo las que se ejecutaban.

En el siguiente ejemplo cambiaremos la disponibilidad del nodo *docker* a *Pause*. Este nodo cuenta con dos tareas desplegadas, invocadas por un servicio que requiere tres réplicas.

```
root@docker:~$ docker service create --name registry --publish 5000:5000 --replicas=3
registry
jxtj9uknr1gjsezi8uodcl7mz
root@docker:~$ docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE           ERROR               PORTS
ib7f1vhri27b        registry.1          registry:latest     docker              Running
Preparing 7 seconds ago
y0210ih2vc4d        registry.3          registry:latest     docker              Running
Preparing 7 seconds ago

root@docker:~$ docker node ps docker2
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE               ERROR               PORTS
24329ftb6bvk        registry.2          registry:latest     docker2             Running
Preparing 16 seconds ago

root@docker:~$ docker node update --availability pause docker

root@docker:~$ docker node ls
ID                           HOSTNAME            STATUS              AVAILABILITY
MANAGER STATUS
e7bb4o42sm9o3h2h2ek4i9pvi    prueba02            Ready               Active
z5jnl56d54vb53a538r1jc23i *  prueba              Ready               Pause
Leader
```

Una vez que cambiamos su disponibilidad escalamos el servicio a diez réplicas para comprobar que no se desplegará ninguna tarea más en el nodo que tiene disponibilidad *Pause*.

```
docker@prueba:~$ docker service scale registry=10
registry scaled to 10

docker@prueba:~$ docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
ib7f1vhri27b        registry.1          registry:latest     prueba              Running
Preparing about a minute ago
y0210ih2vc4d        registry.3          registry:latest     prueba              Running
Preparing about a minute ago
docker@prueba:~$ docker node ps prueba02
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE                ERROR               PORTS
24329ftb6bvk        registry.2          registry:latest     prueba02            Running
Preparing about a minute ago
dqux3t4nz6q6        registry.4          registry:latest     prueba02            Running
Preparing 16 seconds ago
lwbhwqkl8boe        registry.5          registry:latest     prueba02            Running
Preparing 16 seconds ago
urajgkztn036        registry.6          registry:latest     prueba02            Running
Preparing 6 seconds ago
qioxfkzoq3l4        registry.7          registry:latest     prueba02            Running
Preparing 6 seconds ago
n5kbregu92j9        registry.8          registry:latest     prueba02            Running
Preparing 6 seconds ago
9f41fx628d4j        registry.9          registry:latest     prueba02            Running
Preparing 6 seconds ago
98y5ice3xaic        registry.10         registry:latest     prueba02            Running
```

Etiquetar un nodo

Con la opción *update* podemos poner etiquetas a nuestro nodo mediante clave=valor, de manera que sean consultables en la información detallada del nodo a través del parámetro *inspect*.

Esto sirve para añadir especificaciones o información adicional, como puede ser el entorno (desarrollo/producción), el nombre del administrador del nodo o cualquier otro tipo de información que necesitemos almacenar.

Un ejemplo más útil sería añadir una etiqueta a los nodos que queremos que permanezcan siempre activos, tratándolos con un script para eliminar aquellos que no pertenezcan a esa etiqueta.

```
root@docker:~$ docker node update --label-add permanente=si docker
docker


root@docker:~$ docker node inspect docker | grep permanente
                "permanente": "si"
root@docker:~$ docker node inspect docker [
    {
        "ID": "wlc3c0ivyatvwdcoz6zlel3pu",
        "Version": {
            "Index": 61
        },
        "CreatedAt":  "2017-05-22T23:17:10.101776653Z",
        "UpdatedAt":  "2017-05-22T23:39:53.519842362Z",
        "Spec": {
            "Labels": {
                "permanente": "si"
            },
            "Role": "manager",
            "Availability": "drain"
        },
        (...)
]
root@docker:~$ docker node inspect -f {{.Spec.Labels.permanente}} docker
si
```

Para consultar todas las etiquetas de un nodo debemos parsear el JSON con el siguiente comando:
`docker node inspect -f {{.Spec.Labels}} docker`

Eliminar etiqueta de un nodo

Podemos eliminar las etiquetas de un nodo con el parámetro *--label-rm* de *update*

```
root@docker:~$ docker node inspect -f {{.Spec.Labels.permanente}} docker
si
root@docker:~$ docker node update --label-rm permanente docker
docker
root@docker:~$ docker node inspect -f {{.Spec.Labels.permanente}} docker
<no value>
```

GESTIÓN DE SERVICIOS

```
root@docker:~$ docker service

Usage: docker service COMMAND

Manage services

Options:
      --help   Muestra su uso

Commands:
  create      Crea un nuevo servicio
  inspect     Muestra información detallada de uno o más servicios
  ls          Lista los servicios
  ps          Lista las tareas de un servicio
  rm          Elimina uno o más servicios
  scale       Escala uno o múltiples servicios replicados
  update      Actualiza un servicio

Run 'docker service COMMAND --help' for more information on a command.
```

Crear un servicio

La manera más simple de crear un servicio es ejecutando el comando `docker service create alpine ping docker.com` donde *alpine* es la imagen a utilizar y *ping docker.com* un comando que ejecutará.

Esto generará un servicio con un nombre aleatorio, sin réplicas (aunque sí en modo replicado) en un nodo sin especificar.

Sin embargo, lo común es especificar al menos el nombre del servicio que creamos a través del parámetro *--name*

```
root@docker:~$ docker service create --name pingdocker alpine ping docker.com
mzx6srrvjsgdf1iev6fepw8zc
```

De esta manera, podría gestionarse el servicio a través del nombre *pingdocker*.

Crear un servicio con especificaciones

A un servicio se le puede pasar múltiples parámetros antes de crearlo. Por ejemplo, el nombre de host que tendrá el contenedor donde se ejecute las tareas del servicio. En este caso, queremos que sea *contenedor*.

```
root@docker:~$ docker service create --name pingdocker --hostname contenedor --replicas=5
alpine ping docker.com
ko18t7ehjcqavuocbpk89iepz
```

Lanzamos cinco réplicas de manera que encontraremos tareas ejecutándose en cualquiera de los nodos. Podemos comprobar el hostname accediendo al nodo de Docker Swarm y conectándonos con una terminal interactiva con el comando `docker exec -it 07eed65b5573 sh` donde *07eed65b5573* es el identificador del contenedor.

También podemos hacerlo a través de la información detallada ofrecida por el parámetro *inspect*: `docker inspect -f {{.Config.Hostname}} 07eed65b5573`

Crear un servicio en modo replicado o global

Al crear un servicio, por defecto se realiza en modo réplica. Aunque no se haya definido ninguna réplica o se haya establecido una única, por lo que hará posible que un servicio pueda ser escalado.

Aún así, existe otro modo llamado *global* que permite desplegar un servicio con una tarea o réplica en cada nodo de Swarm.

Esto quiere decir que tendremos tantas réplicas como nodos pertenezcan a Docker Swarm. En nuestro caso, que tenemos *docker* y *docker2* tendremos dos réplicas del servicio que lancemos con modo global.

```
root@docker:~$ docker service create --name pingdocker --mode global alpine ping
docker.com
medda7er8vmxu84zd6szqf09y
root@docker:~$ docker service ls
ID            NAME        MODE    REPLICAS  IMAGE
medda7er8vmx  pingdocker  global  2/2       alpine:latest
root@docker:~$ docker service ps pingdocker
ID            NAME                                    IMAGE          NODE      DESIRED STATE
CURRENT STATE           ERROR  PORTS
sfwli7j7uuor  pingdocker.fe7yo3nl30hco6qoshbv7k4ie  alpine:latest  prueba    Running
Running 18 seconds ago
6udirbss3tpt  pingdocker.0tptmeffn3u3qp2v69e2n20v2  alpine:latest  prueba02  Running
Running 18 seconds ago
```

Crear un servicio con exposición de puertos

La exposición de puertos hace posible que se asocie un puerto de un servicio a un puerto de nuestro nodo, siendo posible acceder a él desde fuera de la red establecida.

Para ello, tanto la configuración del contenedor como la configuración de la aplicación que corre dentro del contenedor deberá estar correctamente definida. Es decir, no podemos hacer una asociación de puertos de un servidor web desde 8080 del servicio al 8080 del nodo si el servidor web no está correctamente configurado para servir por el 8080.

Sí podemos hacer que el servidor web (que sirve por el puerto 80) se asocie con el puerto 8080 del servicio de Docker. Pudiendo acceder a través de la dirección IP del nodo y el puerto 8080.

Para ello, utilizamos una imagen de nginx y ejecutamos el siguiente comando: `docker service create --name servidorweb --publish 8080:80 nginx`

```
root@docker:~$ docker service create --name servidorweb --publish 8080:80 nginx
ljxrjcqk27uqppnarf1wvzvfn
root@docker:~$ docker service ps servidorweb
ID            NAME           IMAGE         NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
raa3ac4annqo  servidorweb.1  nginx:latest  docker2  Running        Running 11 minutes ago
root@docker:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
0tptmeffn3u3qp2v69e2n20v2 *  docker    Ready   Active        Leader
fe7yo3nl30hco6qoshbv7k4ie    docker2  Ready   Active root@docker:~$
docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                           SWARM   DOCKER
ERRORS
prueba     *        virtualbox   Running   tcp://192.168.99.100:2376           v17.05.0-ce
prueba02   -        virtualbox   Running   tcp://192.168.99.101:2376           v17.05.0-ce
```

```
root@docker:~$ docker service ps servidorweb
ID            NAME           IMAGE         NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
raa3ac4annqo  servidorweb.1  nginx:latest  docker2  Running        Running 11 minutes ago
root@docker:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
0tptmeffn3u3qp2v69e2n20v2 *  docker    Ready   Active        Leader
fe7yo3nl30hco6qoshbv7k4ie    docker2  Ready   Active root@docker:~$
docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                           SWARM   DOCKER
ERRORS
prueba     *        virtualbox   Running   tcp://192.168.99.100:2376           v17.05.0-ce
prueba02   -        virtualbox   Running   tcp://192.168.99.101:2376           v17.05.0-ce
```

En este caso podemos ver la pantalla de "Welcome to nginx!" accediendo a cualquier nodo de Docker Swarm a través del puerto 8080. Esto es debido al routing mesh que Docker Swarm, un mecanismo que permite que un servicio sea accesible por el mismo puerto en todos los nodos, incluso si el nodo no tiene el servicio desplegado en él.

Crear un servicio con variables de entorno

Al crear un servicio es posible establecer unas variables para todas las tareas de ese servicio. Estas pueden contener múltiples valores como por ejemplo la configuración de un proxy.

```
root@docker:~$ docker service create --name pingdocker --env
HTTP_PROXY=http://proxy.iesgn.org:8000 --env HTTPS_PROXY=http://proxy.iesng.org:8000 alpine
ping docker.com
lkz4ej2r5g2yikk4cw31aso0a
```

Como se ve, es necesario utilizar *--env* para cada variable de entorno que queremos establecer.

Puede consultarse a través de la información detallada que nos devuelve la opción *inspect*.

```
root@docker:~$ docker inspect -f {{.Spec.TaskTemplate.ContainerSpec.Env}} pingdocker
[HTTP_PROXY=http://proxy.iesgn.org:8000 HTTPS_PROXY=http://proxy.iesng.org:8000
```

Crear un servicio con constraints

Al crear un servicio, podemos especificar constraints o restricciones principalmente sobre los nodos donde se ejecutarán las tareas.

Por ejemplo, podemos indicar que un servicio nunca se ejecute tareas en un nodo específico, bien sea por seguridad, por motivos de rendimiento o por cualquier otra razón. O que se ejecute todas las tareas de ese servicio en un mismo nodo.

Para que nunca se ejecute en un nodo específico lo hacemos indicando la negación con *!=* de manera que quedaría un comando así `docker service create --name pingdocker --constraint 'node.hostname != docker' alpine ping docker.com`

Las tareas del servicio *pingdocker* nunca se ejecutarían en el nodo *docker*, aunque este se escalase.

En el siguiente ejemplo se ejecutará un servicio con nombre *pingdocker*, con 15 réplicas en el nodo de Docker Swarm con hostname *docker2*

```
root@docker:~$ docker service create --name pingdocker --constraint 'node.hostname ==
docker2' --replicas=15 alpine ping docker.com
pcxazkpumn7m52xkp1sjd7rhi

root@docker:~$ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
pcxazkpumn7m  pingdocker  replicated  15/15     alpine:latest

root@docker:~$ docker service ps pingdocker
ID            NAME            IMAGE          NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
ut3x76al6dwc  pingdocker.1    alpine:latest  docker2  Running        Running 17 seconds ago
1qijlprxkb97  pingdocker.2    alpine:latest  docker2  Running        Running 17 seconds ago
a0sr68jzt3mt  pingdocker.3    alpine:latest  docker2  Running        Running 18 seconds ago
0cw0ns12cghn  pingdocker.4    alpine:latest  docker2  Running        Running 18 seconds ago
a4whj9frs74s  pingdocker.5    alpine:latest  docker2  Running        Running 17 seconds ago
z7zt6c00ihn7  pingdocker.6    alpine:latest  docker2  Running        Running 17 seconds ago
gkzzciejl2ig  pingdocker.7    alpine:latest  docker2  Running        Running 18 seconds ago
gr1qylnaey0f  pingdocker.8    alpine:latest  docker2  Running        Running 18 seconds ago
khv265lxanqi  pingdocker.9    alpine:latest  docker2  Running        Running 18 seconds ago
3htn0pdqqgtc  pingdocker.10   alpine:latest  docker2  Running        Running 17 seconds ago
tittb54fupnk  pingdocker.11   alpine:latest  docker2  Running        Running 17 seconds ago
k008j0jb42ae  pingdocker.12   alpine:latest  docker2  Running        Running 17 seconds ago
u3qs2zocloo4  pingdocker.13   alpine:latest  docker2  Running        Running 18 seconds ago
rfhvktswbao1  pingdocker.14   alpine:latest  docker2  Running        Running 18 seconds ago
mhi2d84tdpnz  pingdocker.15   alpine:latest  docker2  Running        Running 18 seconds ago

root@docker:~$ docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
0tptmeffn3u3qp2v69e2n20v2 *   prueba    Ready   Active        Leader
fe7yo3nl30hco6qoshbv7k4ie     prueba02  Ready   Ace
```

Podemos comprobar que el servicio está replicado 15/15 en el nodo *docker2*.

Crear un servicio con especificaciones de update

Es posible especificar cómo se comporta un servicio a la hora de realizar una actualización de este. Por ejemplo, podemos definir cuánto tiempo ha de pasar entre actualización de una tarea y otra, o en qué cantidad se realiza (de uno en uno, de cinco en cinco...)

```
root@docker:~$ docker service create --name pingdocker --update-delay 30s --update-
parallelism 2 alpine ping docker.com
jcl81hgsx6vw3vnzh8prtuy61
```

De esta manera se actualizarían dos tareas cada 30 segundos. Es decir, pararía dos tareas del servicio, las iniciaría con la nueva configuración del update y pasado 30 segundos volvería a hacerlo con las otras dos tareas siguientes.

Listar servicios

Con la ejecución de `docker service ls` podremos ver qué servicios se están ejecutando en nuestro Swarm, así como otro tipo de información como el modo (*replicated* o *global*) o el número de réplicas.

```
root@docker:~$ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
syp2nt42zx5t  pingdocker  replicated  3/3       alpine:latest
vj40hbkxzw98  registry    replicated  1/1       registry:latest
```

Con la opción *--filter* podemos filtrar el listado mostrado a través de clave=valor. Puede filtrarse según el id, nombre, modo o labels en caso de haber usado alguno al crear el servicio. En este ejemplo muestro los servicios que están en modo réplica.

```
root@docker:~$ docker service ls --filter mode=replicated
ID            NAME        MODE        REPLICAS  IMAGE
syp2nt42zx5t  pingdocker  replicated  3/3       alpine:latest
vj40hbkxzw98  registry    replicated  1/1       registry:latest
```

Si utilizamos el parámetro *-q* nos mostrará solo el ID del servicio.

```
root@docker:~$ docker service ls -q
syp2nt42zx5tf6zgkryeb4pt6
vj40hbkxzw9832e5wrbmnkkfl
```

Listar tareas de un servicio

Un servicio está compuesto por tareas. Docker Swarm despliega tareas de este servicio, estas tareas se ejecutan en contenedores. Estos contenedores reciben el nombre del servicio y una numeración incremental según el número de réplicas.

Consultamos los servicios activos, en este caso un ping a docker.com con cinco réplicas.

```
root@docker:~$ docker service ls
ID            NAME        MODE        REPLICAS  IMAGE
nheealov3hm4  pingdocker  replicated  5/5       alpine:latest
```

Seguidamente consultamos con `docker service ps pingdocker` las tareas que ejecuta ese servicio.

```
root@docker:~$ docker service ps pingdocker
ID            NAME           IMAGE           NODE      DESIRED STATE  CURRENT
STATE ERROR  PORTS
kpmw7s1j1c07  pingdocker.1   alpine:latest   prueba    Running        Running 39 seconds ago
rhahazoq6c07  pingdocker.2   alpine:latest   prueba02  Running        Running 39 seconds ago
zccmotnfgmq2  pingdocker.3   alpine:latest   prueba02  Running        Running 39 seconds ago
e54lld0du4r9  pingdocker.4   alpine:latest   prueba    Running        Running 39 seconds ago
ilbigwihw9k3  pingdocker.5   alpine:latest   prueba02  Running        Running 39 seconds ago
```

Acceder al contenedor que ejecuta una tarea de un servicio

Aunque acceder a contenedores no es una tarea habitual, es posible que deseéis acceder a un contenedor que ejecuta una tarea.

Para hacerlo, debemos listar las tareas de un servicio con `docker service ps pingdocker`

```
root@docker:~$ docker service ps pingdocker
ID            NAME           IMAGE           NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
zbejdmsdr91x  pingdocker.1   alpine:latest   docker    Running        Running 19 minutes ago
5lvo8xxfdtz6  pingdocker.2   alpine:latest   docker    Running        Running 19 minutes ago
vw0y68kn4d4o  pingdocker.3   alpine:latest   docker2   Running        Running 19 minutes ago
ovgckl6v1cic  pingdocker.4   alpine:latest   docker2   Running        Running 19 minutes ago
n3rcis5t2b90  pingdocker.5   alpine:latest   docker2   Running        Running 19 minutes ago
```

Luego acceder al nodo que ejecuta esa tarea y ejecutar un `docker ps` para ver el contenedor que la ejecuta.

```
root@docker:~$ docker ps
CONTAINER ID        IMAGE
          COMMAND              CREATED             STATUS
83d8dfca01fe
alpine:latest@sha256:0b94d1d1b5eb130dd0253374552445b39470653fb1a1ec2d81490948876e462c   "ping
docker.com"   20 minutes ago      Up 20
minutes pingdocker.3.vw0y68kn4d4o5ffn29pc4fl7o
e397df06d71e
alpine:latest@sha256:0b94d1d1b5eb130dd0253374552445b39470653fb1a1ec2d81490948876e462c
"ping docker.com"   20 minutes ago      Up 20 minutes
pingdocker.4.ovgckl6v1cic6vovu9uqpkgsm
4fb752459551
alpine:latest@sha256:0b94d1d1b5eb130dd0253374552445b39470653fb1a1ec2d81490948876e462c
"ping docker.com"   20 minutes ago      Up 20 minutes
pingdocker.5.n3rcis5t2b90f4pkfy2si79qm
```

Entonces ejecutamos `docker exec -it 83d8dfca01fe sh`

Escalar un servicio

Para escalar un servicio de docker basta con usar el parámetro *scale* y en clave=valor poner el nombre del servicio y el número de réplicas a las que se desea escalar.

```
root@docker:~$ docker service scale pingdocker=15
pingdocker scaled to 15
root@docker:~$ docker service ls
ID            NAME            MODE        REPLICAS  IMAGE
nheealov3hm4  pingdocker      replicated  15/15     alpine:latest
root@docker:~$ docker service ps pingdocker
ID            NAME            IMAGE          NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
kpmw7s1j1c07  pingdocker.1    alpine:latest  docker    Running        Running 11 minutes ago
rhahazoq6c07  pingdocker.2    alpine:latest  docker2   Running        Running 11 minutes ago
zccmotnfgmq2  pingdocker.3    alpine:latest  docker2   Running        Running 11 minutes ago
e54lld0du4r9  pingdocker.4    alpine:latest  docker    Running        Running 11 minutes ago
ilbigwihw9k3  pingdocker.5    alpine:latest  docker2   Running        Running 11 minutes ago
nebsnelo8ydi  pingdocker.6    alpine:latest  docker2   Running        Running 9 seconds ago
jhs9xn3hyge1  pingdocker.7    alpine:latest  docker    Running        Running 9 seconds ago
umkn89bglckt  pingdocker.8    alpine:latest  docker    Running        Running 9 seconds ago
a555lghvuydf  pingdocker.9    alpine:latest  docker    Running        Running 9 seconds ago
pif9dwmfgrgs  pingdocker.10   alpine:latest  docker2   Running        Running 9 seconds ago
w47url63i1pu  pingdocker.11   alpine:latest  docker2   Running        Running 9 seconds ago
w8sxdzbiybe2  pingdocker.12   alpine:latest  docker2   Running        Running 9 seconds ago
zf554xcx58ds  pingdocker.13   alpine:latest  docker    Running        Running 9 seconds ago
r0n712nso4qm  pingdocker.14   alpine:latest  docker    Running        Running 9 seconds ago
keebp9ozgokp  pingdocker.15   alpine:latest  docker2   Running        Running 9 seconds ago
```

Este proceso no podría hacerse con servicios desplegados en modo global.

Actualizar un servicio

Una de las actualizaciones que podemos realizar es el tiempo de espera entre actualización y actualización de cada contenedor. En este caso establecemos esta opción a un minuto.

```
root@docker:~$ docker service update --update-delay 1m pingdocker
pingdocker
```

Al ejecutar cualquier otra actualización del servicio se esperará un minuto para reiniciar la tarea y volverla a lanzar (no necesariamente en el mismo nodo). En este ejemplo ha comenzado con la tarea *pingdocker.2*

```
nano@satellite:~$ docker service update --hostname=container pingdocker
pingdocker
nano@satellite:~$ docker service ps pingdocker
ID            NAME               IMAGE          NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
jw4d1cjlaorw  pingdocker.1       alpine:latest  prueba02  Running        Running 51 seconds
ago
d0s3iyjlc52i  pingdocker       alpine:latest  prueba    Ready          Ready 3 seconds ago
4zcbvbky69r5   \_ pingdocker.2  alpine:latest  prueba02  Shutdown       Running 3 seconds ago
urbot3qyildj  pingdocker.3       alpine:latest  prueba    Running        Running 52 seconds
ago
kco36opobjmx  pingdocker.4       alpine:latest  prueba02  Running        Running 51 seconds
ago
gsbz60km1b3x  pingdocker.5       alpine:latest  prueba    Running        Running 52 seconds
ago
```

Quiere decir que si el servicio requiere de cinco réplicas, habrá cinco tareas y por lo tanto cuatro minutos para que se termine el proceso de actualización, a no ser que se indique una condición de paralelismo. El paralelismo indica cuántas tareas actualizará a la vez y se especifica con *--update-parallelism*

También existe la posibilidad de que el servicio ya esté creado con estas especificaciones de actualización.

Rollback de un servicio

El rollback de un servicio permite deshacer una actualización efectuada en un servicio, siguiendo las mismas condiciones de actualización que en el punto *2.11.8*

Con `docker service update --rollback pingdocker` las tareas de nuestro servicio volverán a ejecutarse en contenedores que tendrán como hostname *contenedor*

```
root@docker:~$ docker service update --rollback pingdocker
pingdocker

root@docker:~$ docker service ps pingdocker
ID             NAME                IMAGE          NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
jfp12srlywll   pingdocker.1        alpine:latest  docker    Ready          Ready 4 seconds ago
jm15zr5fgzt9    \_ pingdocker.1    alpine:latest  docker    Shutdown       Running 4 seconds ago
jw4d1cjlaorw    \_ pingdocker.1    alpine:latest  prueba02  Shutdown       Shutdown 8 minutes
ago
d0s3iyjlc52i   pingdocker.2        alpine:latest  prueba    Running        Running 10 minutes
ago
4zcbvbky69r5    \_ pingdocker.2    alpine:latest  prueba02  Shutdown       Shutdown 10 minutes
ago
nt1kdv1z656p   pingdocker.3        alpine:latest  prueba    Running        Running 6 minutes ago
urbot3qyildj    \_ pingdocker.3    alpine:latest  prueba    Shutdown       Shutdown 6 minutes
ago
nriflgmbjntt   pingdocker.4        alpine:latest  prueba02  Running        Running 7 minutes ago
kco36opobjmx    \_ pingdocker4     alpine:latest  prueba02  Shutdown       Shutdown 7 minutes
ago
xidjz06sgw7l   pingdocker.5        alpine:latest  prueba02  Running        Running 9 minutes ago
bz60km1b3x      \_ pingdocker.5    alpine:latest  prueba    Shutdown       Shutdown 9 minutes ag
```

Al finalizar el rollback tendremos una tarea más por cada réplica.

```
root@docker:~$ docker service ps pingdocker
ID             NAME                IMAGE          NODE      DESIRED STATE  CURRENT STATE
ERROR  PORTS
jfp12srlywll   pingdocker.1        alpine:latest  docker    Running        Running 17 minutes
ago
jm15zr5fgzt9    \_ pingdocker.1    alpine:latest  docker    Shutdown       Shutdown 17 minutes
ago
jw4d1cjlaorw    \_ pingdocker.1    alpine:latest  docker2   Shutdown       Shutdown 25 minutes
ago
h630i36uj66u   pingdocker.2        alpine:latest  docker    Running        Running 12 minutes
ago
d0s3iyjlc52i    \_ pingdocker.2    alpine:latest  docker    Shutdown       Shutdown 12 minutes
ago
4zcbvbky69r5    \_ pingdocker.2    alpine:latest  docker2   Shutdown       Shutdown 28 minutes
ago
33olfu0am21u   pingdocker.3        alpine:latest  docker    Running        Running 15 minutes
ago
nt1kdv1z656p    \_ pingdocker.3    alpine:latest  docker    Shutdown       Shutdown 15 minutes
ago
urbot3qyildj    \_ pingdocker.3    alpine:latest  docker    Shutdown       Shutdown 23 minutes
ago
9liuc0yux01p   pingdocker.4        alpine:latest  docker2   Running        Running 16 minutes
ago
nriflgmbjntt    \_ pingdocker.4    alpine:latest  docker2   Shutdown       Shutdown 16 minutes
ago
kco36opobjmx    \_ pingdocker.4    alpine:latest  docker2   Shutdown       Shutdown 24 minutes
ago
ylu19vgk7iyx   pingdocker.5        alpine:latest  docker2   Running        Running 13 minutes
ago
xidjz06sgw7l    \_ pingdocker.5    alpine:latest  docker2   Shutdown       Shutdown 13 minutes
ago
gsbz60km1b3x    \_ pingdocker.5    alpine:latest  docker    Shutdown       Shutdown 27 minutes
ago
```

Eliminar servicios

De la misma forma que borramos un contenedor, utilizamos la opción *rm* para borrar un servicio. Los servicios no tienen estados (start/stop), por lo que el servicio será borrado aunque se encuentre en ejecución.

```
root@docker:~$ docker service rm pingdocker
pingdocker
```

Para eliminar todos los servicios podemos ejecutar `docker service rm $(docker service ls -q)`

GESTIÓN DE STACKS

Los ***stacks*** en Docker Swarm son definiciones en un archivo de texto en formato YAML de múltiples servicios además de volúmenes, redes de software y secretos. Esta definición de un ***stack*** ejecutado en un ***cluster*** de nodos Docker Swarm permite iniciar múltiples contenedores además de los otros elementos que necesiten para su funcionamiento. Los ***stacks*** son el equivalente para Docker Swarm de los archivos multicontenedor de Docker Compose, y el formato de ambos muy similar.

```
root@docker:~$ docker stack

Usage: docker stack COMMAND

Manage Docker stacks

Options:
      --help   Muestra su uso

Commands:
  deploy      Despliega un nuevo stack o actualiza un stack existente
  ls          Lista los stacks
  ps          Lista las tareas de un stack
  rm          Elimina el stack
  services    Lista los servicio de un stack

Run 'docker stack COMMAND --help' for more information on a command.
```

Desplegar o actualizar un stack

Para desplegar un nuevo stack de servicios en Docker Swarm podemos utilizar el comando `docker stack up -c stackdocker.yml docker` o `docker stack deploy -c stackdocker.yml docker`

```
root@docker:~$ docker stack deploy -c stackdocker.yml docker
Creating network docker_default
Creating network docker_backend
Creating network docker_frontend
Creating service docker_vote
Creating service docker_result
Creating service docker_worker
Creating service docker_visualizer
Creating service docker_redis
Creating service docker_db
```

Si deseamos actualizar el stack con modificaciones en el archivo YML podemos hacerlo ejecutando el mismo comando. Los cambios solo afectarán al servicio modificado.

Listar stacks

Podemos gestionar los stacks que tenemos corriendo en nuestro Docker Swarm con la opción *ls* del comando *docker stack.*

```
root@docker:~$ docker stack ls
NAME    SERVICES
docker  6
```

En este caso son seis servicios los que se están ejecutando a través de este stack.

Listar los servicios de un stack

Para ver los servicios pertenecientes a un stack ejecutamos `docker stack services docker`

```
root@docker:~$ docker stack services docker
ID            NAME               MODE        REPLICAS  IMAGE
3m803gub3fus  docker_redis       replicated  2/2       redis:alpine
8dwvg5n4vxmg  docker_result      replicated  1/1
dockersamples/examplevotingapp_result:before
iynush5rtjdx  docker_worker      replicated  1/1
dockersamples/examplevotingapp_worker:latest
nyq747flbec6  docker_visualizer  replicated  1/1       dockersamples/visualizer:stable
p5dheej36nwt  docker_vote        replicated  2/2
dockersamples/examplevotingapp_vote:before
zf7if0ijl5eh  docker_db          replicated  1/1       postgres:9.4
```

Listar las tareas de un stack

Si deseamos ver las tareas que están siendo ejecutadas a través de un stack de servicios debemos lanzar el comando `docker stack ps docker`

```
root@docker:~$ docker stack ps docker
```

Eliminar un stack

Podemos eliminar un stack de servicios con la opción *rm* de *docker stack*.

```
root@docker:~$ docker stack rm docker
Removing service docker_redis
Removing service docker_result
Removing service docker_worker
Removing service docker_visualizer
Removing service docker_vote
Removing service docker_db
Removing network docker_default
Removing network docker_frontend
Removing network docker_backend
```

Restricciones de colocación con Docker Swarm
(placement constraints)

A veces, es necesario controlar *dónde* se ejecuta un contenedor. Esto puede ser por razones de funcionalidad; por ejemplo, un contenedor que supervisa e informa sobre el estado de Swarm debe ejecutarse en un nodo manager para obtener los datos que necesita, o puede haber requisitos del sistema operativo (¡un contenedor diseñado para ejecutarse en una máquina con Windows no debería implementarse en una máquina con Linux!).

También puede suceder que algunos nodos tienen los recursos necesarios para ejecutar nuestros contenedores, y los más comun son las dependencias de los"volumenes", como base de datos…

```
db:
    image: mysql:5.5
    networks:
      - appdb
    environment:
      - MYSQL_ROOT_PASSWORD=foobar
      - MYSQL_DATABASE=mydb1
    volumes:
      - db-data:/var/lib/mysql
    deploy:
      placement:
        constraints: [node.hostname == docker2.curso.local]
```

Restringir un host por nombre de host funciona, pero está limitado a un solo host. ¿Y si quisieras correr 3 copias de por ejemplo un conetenedor de Tomcat?, necesitamos restringirlo para que se ejecute, justo donde están los archivos de configuración

Podemos definir una *label* y restringirla a través de labels:

```
tomcat:
    image: sweh/test:fake_tomcat
    deploy:
      replicas: 1
      placement:
        constraints: [node.labels.Tomcat == true ]
    volumes:
      - "/myapp/apache/certs:/etc/pki/tls/certs/myapp"
      - "/myapp/apache/logs:/etc/httpd/logs"
      - "/myapp/tomcat/webapps:/usr/local/apache-tomcat-8.5.3/webapps"
      - "/myapp/tomcat/logs:/usr/local/apache-tomcat-8.5.3/logs"
    ports:
      - "8443:443"
```

Por ejemplo en el nodo docker2, tendríamos que crear toda la estructura de directorio que van a contener la configuración, o en cada nodo que contenga las etiquetas:

```
/myapp/apache/certs
/myapp/apache/logs
/myapp/tomcat/webapps
/myapp/tomcat/logs
```

Ahora etiquetamos los nodos con la label Tomcat:

```
#docker node update --label-add Tomcat=true docker2
#docker node inspect --format '{{ .Spec.Labels }}' docker2
```

Desplegamos nuestro stack

```
# docker stack deploy --compose-file=stack.tomcat.yml myapp
```

Ahora veremos si el servicio myapp esta corriendo y tiene que estar en el nodo docker2

```
# docker stack ls
# docker stack ps myapp
```

Si escalamos el servicio, los contenedores solo se crearán en los nodos etiquetados en este caso docker2:

```
# docker service scale myapp_tomcat=2
```

Si ahora etiquetamos con la label Tomcat y creamos la estructura de directorio en otro nodo (docker), cuando escalemos el servicio veremos como se crean los nuevos contenedores en este servidor.

Si eliminamos la label Tomcat, en el nodo docker2, vermos como detiene todos los conetenedores y los inicia en otro servidor que tenga la label Tomcat:

```
#docker node update --label-rm Tomcat docker2
# docker service scale myapp_tomcat=3
```

Tabien podremos colocar nuestros contenedores en nodos manager o en nuestros workers:

```
version: '3.0'
services:
  loadbalancer:
    image: traefik
    command: --docker \
      --docker.swarmmode \
      --docker.watch \
      --web \
      --loglevel=DEBUG
    ports:
      - 80:80
      - 9090:8080
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    deploy:
      restart_policy:
        condition: any
      mode: replicated
      replicas: 1
      update_config:
        delay: 2s
      placement:
         constraints: [node.role == manager]
```

Mas ejemplos:

```
version: 3.2
services:
  foo:
    image: foo/bar
    deploy:
      placement:
        constraints:
          node.role != manager
```

deploy:
mode: replicated
replicas: 2
labels: [APP=VOTING]
placement:
constraints: [node.role == worker]

deploy:
mode: replicated
replicas: 2
labels: [APP=VOTING]
placement:
constraints: [node.role == manager

```
docker service create \
    --name nginx-workers-only \
    --constraint node.role==worker \
    nginx

docker service create \
    --name nginx-east \
    --constraint node.labels.Tomcat \
    nginx

docker service create \
    --name nginx-no-Tomcat \
    --constraint node.labels.Tomcat \
    --constraint node.labels.type!=devel \
    nginx
```

Puede usar estas etiquetas para restringir scheduling de un servicio:

```
# docker service create --name TEST --constraint 'node.role == manager' ...
# docker service create --name TEST --constraint 'node.id ==
# docker service create --name TEST --constraint 'node.hostname != docker01' ...
```

Mmultiples constraints, estaríamos relizando un AND:

```
# docker service create --name TEST --constraint 'node.role == manager' --constraint
'node.hostname != docker01' ...
```

Laboratorio Stack de servicios en un cluster de Docker Swarm

Los ***stacks*** en Docker Swarm son definiciones en un archivo de texto en formato YAML de múltiples servicios además de volúmenes, redes de software y secretos. Esta definición de un ***stack*** ejecutado en un ***cluster*** de nodos Docker Swarm permite iniciar múltiples contenedores además de los otros elementos que necesiten para su funcionamiento. Los ***stacks*** son el equivalente para Docker Swarm de los archivos multicontenedor de Docker Compose, y el formato de ambos muy similar.

Con Docker Compose se pueden definir en un único archivo un conjunto de contenedores que forma un servicio o aplicación y que se lanzan como una unidad. En vez de ejecutar los comandos individuales que inician cada contenedor el archivo en formato yaml de Docker Compose define varios contenedores y al ser un archivo de texto es añadible a un sistema de control de versiones para registrar los cambios. La información del archivo de Docker Compose es la misma que se indicaría en el comando para iniciar un contenedor individual.

El modo “swarm” (enjambre) sencillamente se trata de conectar varias máquinas dentro de un clúster al que se unen. Cada una de las máquinas es un nodo. Por un lado tenemos el administrador y por otro los trabajadores. El administrador es quien va dando las órdenes para que los trabajadores las ejecuten. Los administradores pueden ejecutar los trabajos y tienen también la capacidad de unir máquinas al “swarm”. Los trabajadores no pueden decirle a otra máquina lo que pueden hacer.

Es importante saber que las máquinas pueden ser físicas o virtuales. Al habilitar Docker en modo “swarm” la máquina actual se convierte en administrador.

Un “stack” (pila) es un grupo de servicios interrelacionados, comparten dependencias y se pueden escalar juntos. Los servicios se pueden relacionar entre sí y ejecutarlos en varias máquinas.

https://docs.docker.com/engine/swarm/

https://docs.docker.com/get-started/part5/#add-a-new-service-and-redeploy

Los beneficios de implementar su aplicación como un microservicio superan con creces cualquier complejidad o costo adicional. Alta disponibilidad, redundancia, robustez, costo y facilidad de desarrollo son solo algunas de las ventajas.

Docker ha acelerado significativamente este proceso. Con Docker, puede empaquetar sus aplicaciones en contenedores ligeros y ejecutarlos en cualquier sistema con un Docker Engine. Cuando se lanzó Docker Swarm, trajeron orquestación nativa a nuestros clusters de producción. Las aplicaciones sin estado se ajustan perfectamente a este modelo de implementación; sin embargo, ¿qué pasa con los servicios con estado?.

La configuración de aplicaciónes con estado, para trabajar con balanceadores de carga en modo de alta disponibilidad solía ser bastante compleja. Necesitaba apuntar los balanceadores de carga a un conjunto de proxies inversos. Estos proxies necesitan almacenar en caché cookies u otros metadatos para determinar cómo enrutar el tráfico de las conexiones web a los contenedores que brindan sus servicios. Todo esto era bastante complicado y costoso, hasta que llegaron Traefik y Docker Swarm.

Traefik es un balanceador de carga/proxy inverso que está diseñado para su uso con microservicios. Es compatible con una gran cantidad de tecnologías. En este laboratorio, veremos cómo usar Traefik con Docker Swarm para servir un blog de WordPress en modo de alta disponibilidad con sticky session.

Las sesiones fijas básicamente significan que el proxy inverso recordará qué conexiones están asociadas con cada servidor/contenedor y continuará enrutando esos mensajes HTTP. Esto es extremadamente útil para las aplicaciones que requieren que inicie sesión.

El primer paso que realizar es **crear una red overlay para swarm**, a través de la cual, configuraremos nuestros contenedores, en nuetro nodo Leader :

 # docker network create -d overlay net

El formador suministrara todos los archivos yml para realizar este laboratorio

Tendremos que aseguaranos que nuestros stacks tengan configurado correctamente las (restricciones de colocación),placement constraints.

Laboratorio 1 Crear nuestro primer stack en Swarm

En este laboratorio crearemos nuestro primer stack y veremos como desplegarlo y escalarlo, solo tenemos un servicio llamado cluster, tendremos que asegurarnos que en nuestro nodos worker no este listando el puerto 80.

docker-compose.yml

```
version: "3"
services:
  cluster:
   image: russmckendrick/cluster
   ports:
     - "80:80"
   deploy:
     replicas: 6
     restart_policy:
       condition: on-failure
     placement:
       constraints:
         - node.role == worker
```

Desplegamos el stack:

```
[root@docker ~]# docker stack deploy --compose-file=docker-compose.yml cluster
```

Chequeamos es estatus del stack

```
# docker stack ls
```

Esto mostrará que se ha creado un solo servicio. Puede obtener detalles del servicio creado por el stack ejecutando este comando:

```
#docker stack services cluster
```

Para ver donde están corriendo los contenedores dentro del stack:

```
#docker stack ps cluster
```

Para escalar el stack:

```
# docker service scale cluster_cluster=2
```

Para ver donde están corriendo los contenedores dentro del stack:

```
#docker stack ps cluster
```

Eliminar el stack:

```
#docker stack rm cluster
```

Laboratorio 2 Creacio de stacks en Swarm

A continuación, creamos el archivo `wordpress.yml:`

`#vi wordpress.yml`

```
version: '3'

services:
  db:
    image: mysql:5.7
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
        max_attempts: 3
      placement:
        constraints: [node.hostname == docker2]
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - net
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
        max_attempts: 3
    volumes:
      - wordpress_data:/var/www/html
    networks:
      - net
    ports:
      - "8001:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress

volumes:
```

```
  db_data:
  wordpress_data:


networks:
  net:
    external: true
```

Esto es un archivo Docker Compose. Este archivo especifica la configuración para un conjunto de contenedores que implementará como un microservicio. Lo principal a mirar es la sección de "réplicas" debajo de "wordpress". En este archivo de configuración, definimos que el frontend de WordPress tendrá un total de 3 réplicas, es decir, 3 contenedores diferentes servirán el código PHP que alimenta WordPress.

Ahora desplegamos la aplicación en docker swarm:

 # docker stack deploy --compose-file=wordpress.yml wordpress

 # docker stack ls

NAME SERVICES ORCHESTRATOR

wordpress-sticky 2 Swarm

 # docker service ls

ID NAME MODE REPLICAS IMAGE PORTS
o7z2gsxmjnxf wordpress-sticky_db replicated 1/1 mysql:5.7
esdjh0kdqe0d wordpress-sticky_wordpress replicated 3/3 wordpress:latest *:30002->80/tcp

Si ahora apuntamos a un nodo **al puerto 8001**, podríamos realizar la instalación de nuestro wordpress, eneste laboratorio solo nos aseguramos que tengamos la instalación.

Por defecto, Docker Swarm utiliza el enrutamiento basado en round robin cada vez que accede al blog. Debido a esto, nuestro navegador utiliza un contenedor diferente para cada solicitud. Después de iniciar sesión en el primer contenedor, su tráfico se enruta al segundo, al cual no ha iniciado sesión. Y luego, el tercero, en el que no ha iniciado sesión. Después de que haya iniciado sesión en los 3, entonces toda funcionaria bien. Esto es útil para aplicaciones donde no se inicia sesión, pero para aplicaciones donde se realice inicio de sesiones, se tendremos que tener configurado sticky session.

Ahora eliminamos el stack desplegado en docker swarm:

```
# docker stack rm wordpress
```

Ahora implementaremos WordPress pero utilizaremos Traefik para servir como un proxy inverso, y configuraremos sticky session.

Crea un archivo llamado **traefik.yml** y pega el siguiente contenido:

```yaml
version: '3.0'

services:
  loadbalancer:
    image: traefik
    command: --docker \
      --docker.swarmmode \
      --docker.watch \
      --web \
      --loglevel=DEBUG
    ports:
      - 80:80
      - 9090:8080
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    deploy:
      restart_policy:
        condition: any
      mode: replicated
      replicas: 1
      update_config:
        delay: 2s
      placement:
         constraints: [node.role == manager]
    networks:
      - net

networks:
    net:
      external: true
```

Desplegamos nuestro balanceador traefick:

```
# docker stack deploy --compose-file=traefik.yml traefik
```

```
# docker service  ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
b0sfe2fmmt5p	traefik_loadbalancer	replicated	1/1	traefik:latest	*:80->80/tcp, *:9090->8080/tcp

http://192.168.1.150:9090/dashboard/

Ahora que hemos implementado Traefik, implementaremos WordPress con una configuración ligeramente modificada.

Cree el archivo wordpress-sticky.yml con el siguiente contenido:

```
version: "3.1"

services:
  db:
    image: mysql:5.7
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
        max_attempts: 3
      placement:
        constraints: [node.hostname == docker2]

    volumes:
      - db_data:/var/lib/mysql
    networks:
      - net
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
```

```
    deploy:
      mode: replicated
      replicas: 3
      restart_policy:
        condition: on-failure
        max_attempts: 3
      placement:
        constraints: [node.hostname == docker2]
      update_config:
        delay: 2s
      labels:
        - "traefik.docker.network=net"
        - "traefik.port=80"
        - "traefik.frontend.rule=PathPrefix:/"
        - "traefik.backend.loadbalancer.sticky=true"
        - "traefik.backend=wp"
        - "traefik.frontend.rule=Host:wp.curso.local"

    volumes:
      - wordpress_data:/var/www/html
    networks:
      - net
    ports:
      - "80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress

networks:
  net:
    external: true

volumes:
  db_data:
  wordpress_data:
```

En el fichero de configuración estamos obligando a los contenedores de Wordpress, ejecutarse en los nodos worker.

Ademas es importante la configuración para hacer pasar wordpres a través de traefik:

labels:

- "traefik.docker.network=net"
- "traefik.port=80"
- "traefik.frontend.rule=PathPrefix:/"
- "traefik.backend.loadbalancer.sticky=true"
- "traefik.backend=wp"
- "traefik.frontend.rule=Host:wp.curso.local"

https://docs.traefik.io/configuration/backends/docker/

Desplegamos nuestro Wordpress con sticky sesión a través de traefik:

```
# docker stack deploy --compose-file=wordpress-sticky.yml wordpress-sticky
```

docker stack ls

NAME SERVICES ORCHESTRATOR

traefik 1 Swarm

wordpress-sticky 2 Swarm

docker service ls

ID NAME MODE REPLICAS IMAGE PORTS

b0sfe2fmmt5p traefik_loadbalancer replicated 1/1 traefik:latest *:80->80/tcp, *:9090->8080/tcp

o7z2gsxmjnxf wordpress-sticky_db replicated 1/1 mysql:5.7

esdjh0kdqe0d wordpress-sticky_wordpress replicated 3/3 wordpress:latest *:30002->80/tcp

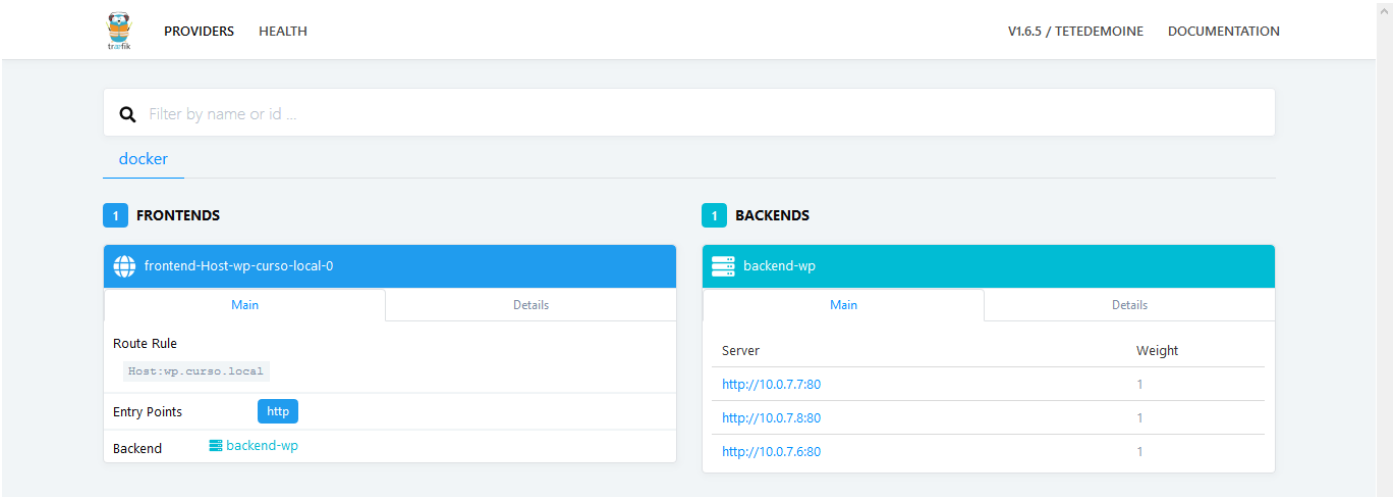
Realizamos la instalación de nuestro Wordpress y nos aseguramos la cuenta de administración y el password, lo necesitaremos para comprobar la configurarion de sticky session

Resolveremos wp.curso.local, en el archivo hosts de nuestro puesto de trabajo.

http://wp.curso.local/

Esta seria la configuración que veriamos en nuestro traefik:

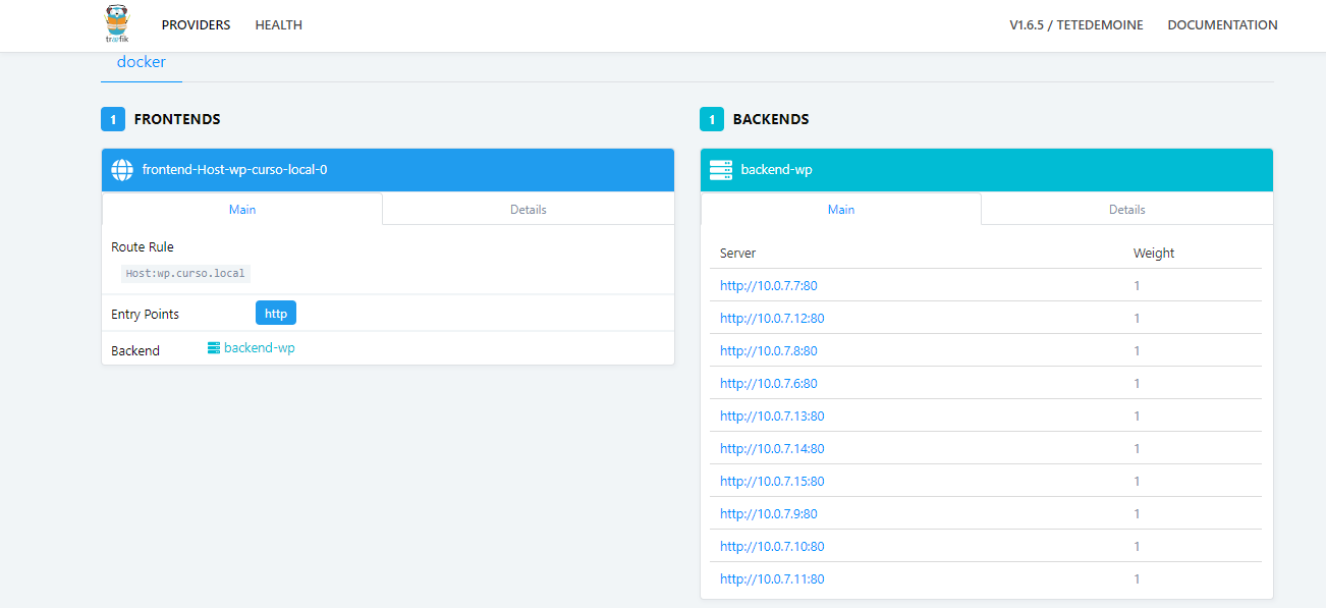
http://192.168.1.150:9090/dashboard/



Ahora cuando establecemos conexión contra un contenedor y realicemos loguin en wordpress, traefik, siempre nos mantendrá la sesión contra ese contenedor.

Ahora podemos escalar nuestro servicio de wordpress:

docker service scale wordpress-sticky_wordpress=10



Si ahora iniciamos sesión, y navegamos dentro de los menus de admin, tendremos que comprobar que la sesion siempre será persistida a través de traefik:

http://wp.curso.local/wp-login.php

Laboratorio Rolling updates with Docker Swarm

https://docs.docker.com/engine/reference/commandline/service/

En este laboratorio veremos como trabaja docker swarm, para realizar actualizaciones en caliente, lanzaremos un servicio llamado **nginx-lab**, con la imagen nbrown/nginxhello:1.12.1, lo escalaremos a 10 contenedores y a continuación realizaremos un Rolling update a la versión nbrown/nginxhello:1.13.5

nbrown/nginxhello:1.12.1

nbrown/nginxhello:1.13.5

Es posible especificar cómo se comporta un servicio a la hora de realizar una actualización de este. Por ejemplo, podemos definir cuánto tiempo ha de pasar entre actualización de una tarea y otra, o en qué cantidad se realiza (de uno en uno, de cinco en cinco...).

Comenzamos el laboratorio:

Con este comando, estamos creado un servicio llamado **nginx-lab**, que tine 4 replicas, estamos publicando el puerto 8080 para llegar a los conenedores.

Para configura la política de actualización progresiva en el momento de la implementación del servicio.

El **`--update-delay`** configura el tiempo de demora entre las actualizaciones de una tarea de servicio o conjuntos de tareas. Puede describir el tiempo `T` como una combinación del número de segundos `Ts` , minutos `Tm` u horas `Th` . Entonces, `10m30s` indica un retraso de 10 minutos y 30 segundos.

Por defecto, el planificador actualiza 1 tarea a la vez. Puede pasar el **`--update-parallelism`** para configurar el número máximo de tareas de servicio que el orquestador actualiza simultáneamente.

De forma predeterminada, cuando una actualización de una tarea individual devuelve un estado de **RUNNING** , el orquestador programa otra tarea para actualizar hasta que se actualicen todas las tareas. Si, en cualquier momento durante una actualización, una tarea devuelve **FAILED** , el orquestador pausa la actualización. Puede controlar el comportamiento utilizando el **`--update-failure-action`** para docker service update **`--update-failure-action,`** docker service create o docker service update .

#docker service create --replicas 4 --name nginx-lab --publish published=8080,target=80 --update-delay 30s --update-parallelism 2 nbrown/nginxhello:1.12.1

[root@docker ~]# **docker service ls**

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
ovc0yf0brjmt	nginx-lab	replicated	4/4	nbrown/nginxhello:1.12.1	*:8080->80/tcp

[root@docker ~]# **docker service inspect --pretty nginx-lab**

http://192.168.1.150:8080/

Si actualizamos el navegador podemos ver como balanceamos entre los distintos contenedore.

Hello!

Hostname: d50dd2cc4152

IP Address: 10.255.0.11:80

Version: 1.12.1

Actualizaciones del servicio

Vamos a actualizar nuestro servicio y ver como es el proceso de actualización de las versiones de los contenedores que se ejecutan en el servicio.

- –replicas: es el número de tareas
- –name: es el nombre del servicio
- –update-delay: es el tiempo que transcurre entre la actualziación de cada una de las tareas.
- **nbrown/nginxhello:1.13.5** es la imagen que vamos a utilizar

Por defecto, se actualiza una tarea cada vez. Podemos modificar este comportamiento con el parámetro –update-parallelism para indicar el número máximo de tareas que se pueden ejecutar de forma simultánea.

Ahora podemos actualizar la imagen del contenedor. El administrador de swarm aplica la actualización a los nodos de acuerdo con la política UpdateConfig :

[root@docker ~]# **docker service update --image nbrown/nginxhello:1.13.5 nginx-lab**

Durante la ejecución podemos ver el estado de las tareas:

[root@docker ~]# **docker service ps nginx-lab**

[root@docker ~]# **docker service inspect --pretty nginx-lab**

[root@docker ~]# **docker service ls**

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
ovc0yf0brjmt	nginx-lab	replicated	4/4	nbrown/nginxhello:1.13.5	*:8080->80/tcp

El orquestador aplica actualizaciones continuas de la siguiente manera por defecto:

- Detener la primera tarea.
- Programa actualización para la tarea detenida.
- Inicia el contenedor para la tarea actualizada.
- Si la actualización de una tarea devuelve **`RUNNING`** , espere el período de retraso especificado y luego comience la siguiente tarea.
- Si, en cualquier momento durante la actualización, una tarea devuelve **`FAILED`** , pausa la actualización.

http://192.168.1.153:8080/

Podemos comprobar como se han actilizado todas las versiones de nuestros contenedores, a la nueva versión.

Hello!

Hostname: 6eda25056e6b

IP Address: 10.255.0.16:80

Version: 1.13.5

Rollback de un servicio

El rollback de un servicio permite deshacer una actualización efectuada en un servicio, siguiendo las mismas condiciones de actualización que en el punto *de creación del servicio:*

[root@docker ~]# **docker service update --rollback nginx-lab**

[root@docker ~]# **docker service ps nginx-lab**

[root@docker ~]# **docker service inspect --pretty nginx-lab**

https://docs.docker.com/engine/reference/commandline/service_rollback/

Ahora podemos observar que estamos en la versión 1.12.1 de los contenedores:

http://192.168.1.153:8080/

Hello!

Hostname: 4a1cef0a1dc4

IP Address: 10.255.0.19:80

Version: 1.12.1

Eliminar servicios

De la misma forma que borramos un contenedor, utilizamos la opción *rm* para borrar un servicio. Los servicios no tienen estados (start/stop), por lo que el servicio será borrado, aunque se encuentre en ejecución.

[root@docker ~]# **docker service rm nginx-lab**

Para eliminar todos los servicios podemos ejecutar:

[root@docker ~]# docker service rm $(docker service ls -q)

Si queremos pasar el servicio de este laboratorio a través de traefick:

Es muy importante averiguar en que red de docker hemos lanzado traefick, en este laboratorio la red de traefick esta en la red llamada net.

[root@docker ~]# **docker network ls**

NETWORK ID	NAME	DRIVER	SCOPE
4fb50b5b3af8	bridge	bridge	local
df1d8451425f	docker_gwbridge	bridge	local
1127cf3cf11a	host	host	local
rta51vesh3qv	ingress	overlay	swarm
lrh2strkkjxb	**net**	**overlay**	**swarm**
6f2af428843d	none	null	local

Tendremos que asegurarnos que tenemos lanzato traefick en nuestro cluster de swarm como servicio:

[root@docker ~]# **docker service ls**

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
awgdikespr75	traefik_loadbalancer	replicated	1/1	traefik:latest	*:80->80/tcp, *:9090->8080/tcp

[root@docker ~]# docker service create \

--replicas 3 \

--name nginx-lab \

--update-delay 30s \

--update-parallelism 2 \

--label traefik.port=80 \

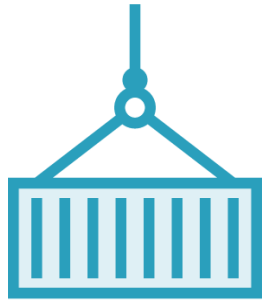
**--network net **

--label traefik.frontend.rule=Host:weblocal.curso.local \

nbrown/nginxhello:1.12.1

[root@docker ~]# **docker service inspect --pretty nginx-lab**

http://weblocal.curso.local/



Hello!

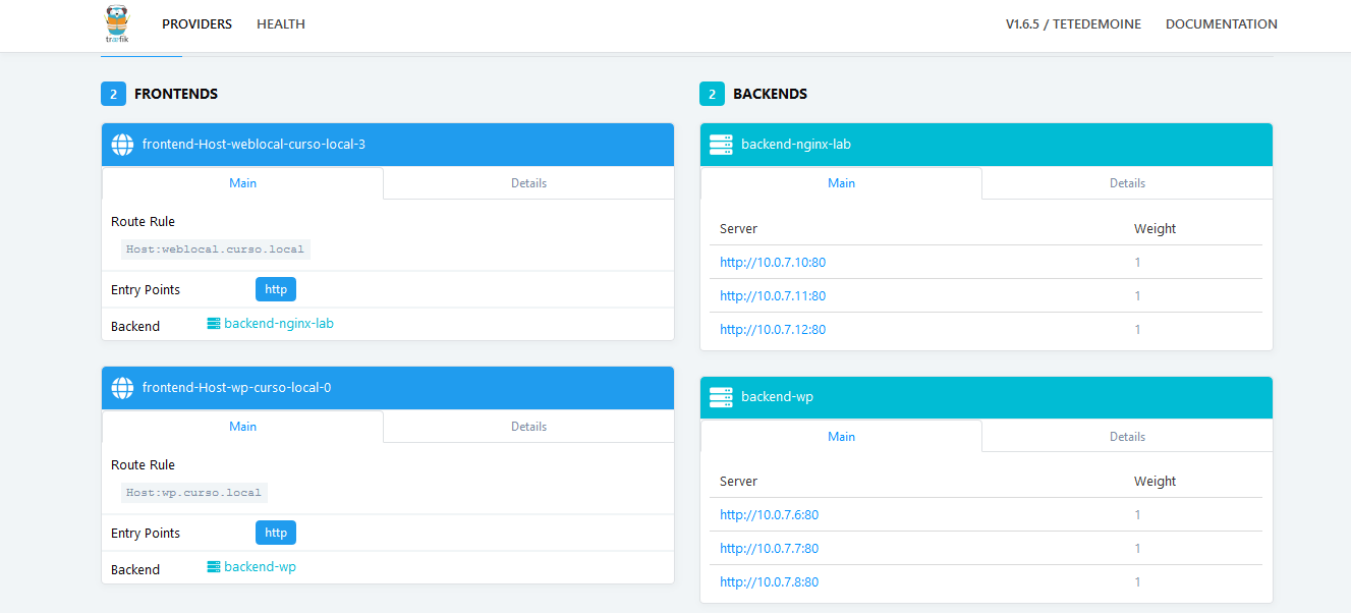
Hostname: 6be220fcc49b

IP Address: 10.0.7.10:80

Version: 1.12.1

Ahora podríamos realizar el update de los contenedores y su vuelta a tras, y comprobarlo todo a través de nuestro balanceador traefick:

http://192.168.1.150:9090/dashboard/



Tambien podríamos realizar el laboratorio a través de **Portainer**.

Docker Secrets

https://docs.docker.com/engine/swarm/secrets/

Los contenedores de Docker necesitan acceder a algunos datos sensibles desde el punto de vista de la seguridad como usuarios y contraseñas, certificados SSL, claves privadas SSH o cualquier otra información de acceso restringido. Algunos de estos datos en Docker se proporcionan mediante variables de entorno al lanzar los contenedores, esto es inseguro ya que al hacer un listado de los procesos con sus parámetros de invocación los relativos a Docker mostrarán esta información, lo que es un posible problema de seguridad.

Con Docker Secrets se puede gestionar esta información que se necesita en tiempo de ejecución pero que no se quiere almacenar en la imagen de Docker o en el repositorio de código fuente. Algunos ejemplos de información sensible son:

- Nombres de usuario y contraseñas.
- Certificados TLS y claves.
- Claves SSH.
- Otra información sensible como el nombre de una base de datos o el nombre de un servidor interno.

Los **secretos** son objetos clave-valor que se almacenan en la base de datos distribuida del **cluster de swarm**. Podremos usar cadenas de caracteres y ficheros; lo que nos lleva rápidamente a observar que tenemos una nueva forma muy **sencilla** de configurar nuestros servicios en el cluster. Hasta ahora, crear un servicio sobre swarm suponía tener la configuración del mismo dentro de la imagen, disponible en todos los hosts de forma local o bien montada mediante almacenamiento de red (por ejemplo usando NFS). Pero los secretos pueden contener ficheros por lo que podemos usarlos para gestionar de forma sencilla las configuraciones de los servicios dado que la información estará disponible en todos los hosts que ejecutan alguna tarea del servicio.

Los secretos de Docker se proporcionan a los contenedores que los necesitan y se transmiten de forma cifrada al nodo en el que se ejecuten. Los secretos se montan en el sistema de archivos en la ruta ***/run/secrets/<secret_name>*** de forma descifrada al que el servicio del contenedor puede acceder.

Cómo funcionan los secretos de Docker

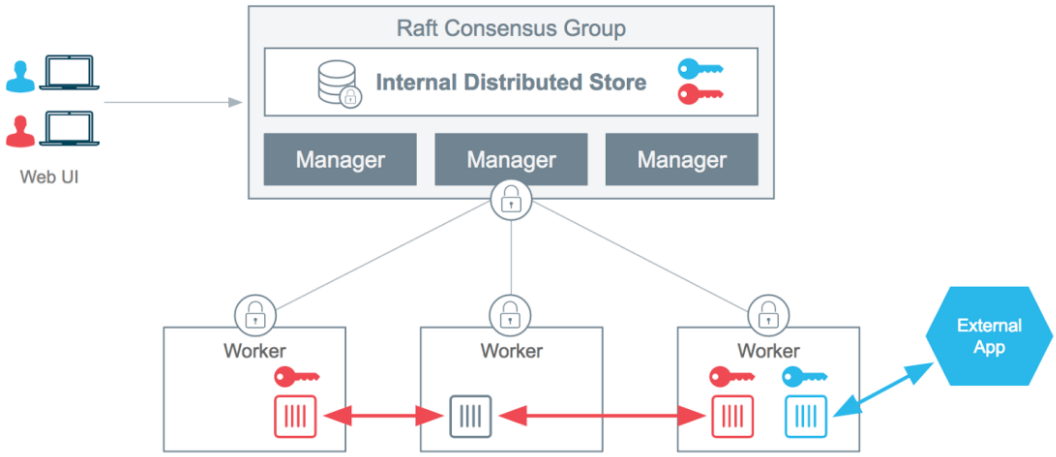
A partir de la versión 1.13, de Docker se pueden usar Docker Secrets en un clúster Swarm. Los managers en Docker Swarm actúan como una delegación autorizada para coordinar la administración de secretos.

Los secretos pueden ser contraseñas, claves, certificados, datos confidenciales del entorno o cualquier otro dato personalizado que un desarrollador quiera proteger, por ejemplo, el nombre de una base de datos, el nombre de usuario de administrador, etc.

Docker Secrets solo está disponible en el modo Swarm, por lo que los contenedores independientes no pueden usar esta función. El modo swarm le permite administrar de forma centralizada los datos y mensajes confidenciales mediante el cifrado y la transmisión de forma segura y solo a los contenedores que necesitan acceder a ellos (el principio de privilegio mínimo).

Cuando un usuario agrega un nuevo secreto a un clúster Swarm, este secreto se envía a un administrador mediante una conexión TLS.

Nota: **TLS es un protocolo criptográfico que proporciona seguridad de comunicaciones a través de una red al proporcionar encriptación de comunicaciones, privacidad e integridad de los datos.**



Para que todos los managers tengan conocimiento de un nuevo secreto creado, cuando un nodo manager recibe el secreto, lo guarda en un almacen **Raft** con una clave de 256 bits.

Nota:

Los nodos de tipo **manager** utilizan Raft para elegir el lider del cluster

https://docs.docker.com/engine/swarm/raft/

https://es.wikipedia.org/wiki/Raft

Algunos comandos para manejar los secretos son los siguientes:

- *docker secret create secreto*: crea un secreto.
- *docker secret inspect secreto*: muestra los detalles de un secreto.
- *docker secret ls*: lista los secretos creados.
- *docker secret rm secreto*: elimina un secreto.
- Se usa el parámetro *–secret* para *docker service create* y *–secret-add* y *–secret-rm flags* para *docker service update*.

Usando un *stack* de servicios con un archivo de Docker Compose en la sección *secrets* de los servicios se indica cuales usa, en la sección *secrets* se definen los secretos de los servicios con sus nombres y su contenido referenciando archivos que pueden ser binarios o de text no superior a 500 KiB.

Laboratorio Docker Secrets

En este laboratorio veremos como configurar docker secret, para la configuración de una base de datos postgres, pasandole la base de datos, usuario y password a través de docker secrets.

Los secretos de Docker ofrecen una forma segura de almacenar información confidencial, como nombres de usuario, contraseñas e incluso archivos como certificados autofirmados.

Antes de comenzar a usar secretos, veamos las desventajas de no usarlos. A continuación se muestra un archivo de compose con la definición de un servicio de Postgres y adminer *(un cliente de base de datos)* :

```
version: '3.1'
services:
  db:
    image: postgres
    environment:
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: 00000000
      POSTGRES_DB: mydatabase
  adminer:
    image: adminer
    ports:
     - 8080:8080
```

Hemos suministrado el nombre de usuario, contraseña y nombre de base de datos para el servicio de Postgres mediante el establecimiento de la variables de entorno **POSTGRES_USER** , **POSTGRES_PASSWORD** , **POSTGRES_DB, como podemos observar las variables están en texto plano y todo el mundo las puede ver.** .

https://hub.docker.com/_/postgres/

Ahora veremos como utilizar los secretos en Swarm:

Creamos un secreto llamado **pg_user** . El guión "-" al final del comando es importante, le permite al comando saber que los datos del secreto se están tomando de la entrada estándar.

#echo "myuser" | docker secret create pg_user -

Para ver el secreto, escriba el siguiente comando:

docker secret ls

ID	NAME	DRIVER	CREATED	UPDATED
emu7v5otmnwbhk3dgb0rknoz1	pg_user		18 hours ago	18 hours ago

Crearemos los secretos restantes para la contraseña y el nombre de la base de datos:

#echo "00000000" | docker secret create pg_password -

#echo "mydatabase" | docker secret create pg_database -

docker secret ls

ID	NAME	DRIVER	CREATED	UPDATED
r15vz6z9vl1jgyqxi5lbqjbs9	mynginxconfig		24 hours ago	24 hours ago
yqwa6aik4zmvumrzpv8vfe2ig	nginx-config-v1		23 hours ago	23 hours ago
dor03nrej97h2xxdcrldn11qx	pg_database		18 hours ago	18 hours ago
jxc9b4rbazowih4dfpmgtlnem	pg_password		18 hours ago	18 hours ago
emu7v5otmnwbhk3dgb0rknoz1	pg_user		18 hours ago	18 hours ago

Ahora modificaremos el archivo de compose, para que pueda usuar los secretos:

vi postgres-secrets.yml

```
version: '3.1'
services:
    db:
        image: postgres
        restart: always
        environment:
            POSTGRES_USER_FILE: /run/secrets/pg_user
            POSTGRES_PASSWORD_FILE: /run/secrets/pg_password
            POSTGRES_DB_FILE: /run/secrets/pg_database
        secrets:
           - pg_password
           - pg_user
           - pg_database
    adminer:
        image: adminer
        ports:
         - 8080:8080
secrets:
  pg_user:
    external: true
  pg_password:
    external: true
  pg_database:
    external: true
```

Para que nuestros secretos funcionen, hemos argregado, las variables de entorno que utilizamos anteriormente han sido modificadas con el sufijo " **_FILE".**

La ruta al secreto también se especifica.

```
POSTGRES_USER_FILE: /run/secrets/pg_user
POSTGRES_PASSWORD_FILE: /run/secrets/pg_password
POSTGRES_DB_FILE: /run/secrets/pg_database
```

Los secretos de Docker se almacenan en archivos en la carpeta **/run/secrets** del contenedor. Por eso tenemos que especificar nuevas variables de entorno para leer los secretos almacenados en estos archivos.

Nota importante: no todas las imágenes tienen variables de entorno compatibles con secrets. En muchos casos tendrás que modificar la definición de imágenes (Dockerfile) para leer los secretos.

En segundo lugar, hemos especificamos el nombre de los secretos que está utilizando el servicio:

```
secrets:
- pg_password
- pg_user
- pg_database
```

Por último, indicamos que los secretos son externos:

```
secrets:
 pg_user:
   external: true
 pg_password:
   external: true
 pg_database:
   external: true
```

Ahora desplegamos el servicio en swarm:

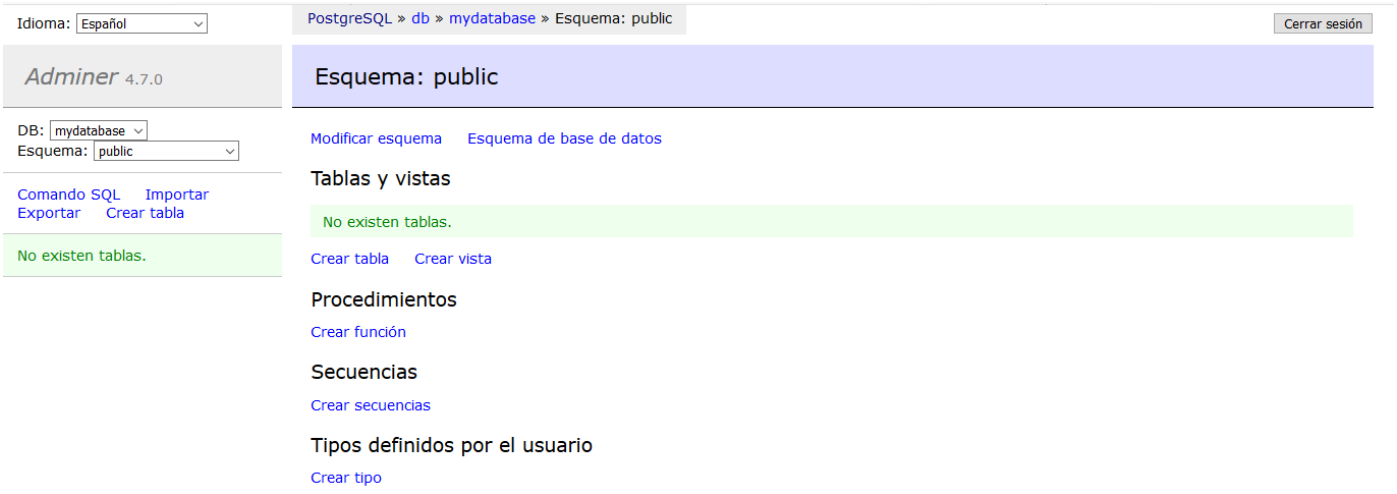
#docker stack deploy -c postgres-secrets.yml postgres

docker service ls

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
j3gsk2uj86lc	postgres_adminer	replicated	1/1	adminer:latest	*:8080->8080/tcp
o7h9qllylc89	postgres_db	replicated	1/1	postgres:latest	

Ahora nos conectamos a nuestro adminer y configuramos la conexión a la base de datos, con los datos configurados en el sercreto:

http://192.168.1.150:8080/



Laboratorio Docker Secrets con MySQL en Swarm

En este laboratorio utilizaremo Docker Secrets, lo que realizaremos sera almacenar nuestras contraseñas de MySQL en Secretos, que se pasarán a nuestros contenedores, de modo que no utilicemos contraseñas de texto plano en nuestros archivos Compose.

Haremos que el servicio MySQL sea persistente al establecer una restricción para que solo se ejecute en el nodo Manager, ya que crearemos la ruta del volumen en el host y luego mapearemos el host al contenedor para que el contenedor pueda tener datos persistentes. También crearemos secretos para nuestro Servicio MySQL para que no expongamos ninguna contraseña en texto plano en nuestro archivo de compose.

Archivo Docker Compose:

mysql-secrets.yml

```
version: '3.3'

services:
  db:
    image: mysql
    secrets:
      - db_root_password
      - db_dba_password
    deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager]
      resources:
        reservations:
          memory: 128M
        limits:
          memory: 256M
    ports:
      - 3306:3306
    environment:
      MYSQL_USER: dba
      MYSQL_DATABASE: mydb
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_PASSWORD_FILE: /run/secrets/db_dba_password
    networks:
      - appnet
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - type: bind
        source: /opt/docker/volumes/mysql
        target: /var/lib/mysql

  adminer:
    image: adminer
    ports:
      - 8080:8080
    networks:
      - appnet

secrets:
  db_root_password:
    external: true
  db_dba_password:
    external: true

networks:
  appnet:
    external: true
```

Para que nuestros secretos funcionen, hemos agregado, las variables de entorno de la imagen de Mysql han sido modificadas con el sufijo " _FILE".

```
MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
MYSQL_PASSWORD_FILE: /run/secrets/db_dba_password
```

Dependencias:

Como especificamos nuestros secretos y redes como recursos externos, debe existir antes de implementar nuestro stack También necesitamos crear el directorio para persistir nuestros datos mysql, ya que los datos se asignarán desde nuestro host a nuestro contenedor.

Creamos la network Overlay:

```
#docker network create --driver overlay appnet
```

Creamos los secretos, para el pass del root y de la base de datos:

```
#echo "00000000" | docker secret create db_root_password -
#echo "00000000" | docker secret create db_dba_password -
```

Listamos los secretos:

```
#docker secret ls
```

```
ID                  NAME                DRIVER              CREATED             UPDATED
grblogvsr0dot1hk0x8k7t880   db_dba_password                 10 seconds ago      10 seconds ago
vpm5ed4z8xmk2i2zhjixmg34x   db_root_password                12 seconds ago      12 seconds ago
```

Inspeccione el secreto, para que podamos ver que no hay valores expuestos:

```
# docker secret inspect db_root_password
[
  {
    "ID": "vpm5ed4z8xmk2i2zhjixmg34x",
    "Version": {
      "Index": 358
    },
    "CreatedAt": "2019-01-19T10:08:22.31466702Z",
    "UpdatedAt": "2019-01-19T10:08:22.31466702Z",
    "Spec": {
      "Name": "db_root_password",
```

```
        "Labels": {}
    }
  }
]
```

Creamos el directorio para persistir los datos de MySQL:

```
#mkdir -p /opt/docker/volumes/mysql
```

Deploy the stack:

```
#docker stack deploy -c mysql-secrets.yml  apps

# docker stack ps apps
```

Connect to MySQL:

El contenido del secreto estará en nuestro contenedor /run/secrets/

```
#  docker exec -it $(docker ps -f name=apps_db -q) ls /run/secrets/

db_dba_password  db_root_password
```

Ver el contenido db_root_password:

```
# docker exec -it $(docker ps -f name=apps_db -q) cat
/run/secrets/db_root_password
```

Conectando a nuestro contenedor de MySQL:

```
# docker exec -it $(docker ps -f name=apps_db -q) mysql -u root -p

Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.7.20 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mydb               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
```

Ahora accedermos a través de adminer WebUI:

http://192.168.1.150:8080

Persistencia de datos

En este punto del laboratorio comprobaremos la persistencia de datos de nuestro contenedor de mysql, crearemos una base de datos llamada **laboratorio**:

```
# docker exec -it $(docker ps -f name=apps_db -q) mysql -u root -p
```

mysql> **create database laboratorio;**

Query OK, 1 row affected (0.00 sec)

mysql> **show databases;**

+--------------------+

| Database |

+--------------------+

| information_schema |

| **laboratorio** |

| mydb |

| mysql |

| performance_schema |

| sys |

+--------------------+

6 rows in set (0.01 sec)

Verificamos el nombre de host de nuestro contenedor, antes de que eliminemos el contenedor:

docker exec -it $(docker ps -f name=apps_db -q) hostname

cb8766502c23

Matamos el contenedor:

docker kill $(docker ps -f name=apps_db -q)

Ahora verificamos el estado del contenedor y comprobamos su nombre, tendremos que ver que el contenedor seara uno nuevo, con un nombre diferente:

```
# docker service ls -f name=apps_db
ID            NAME        MODE          REPLICAS      IMAGE          PORTS
wvz0irstif7d  apps_db     replicated    1/1           mysql:5.7      *:3306->3306/tcp

#docker service ls -f name=apps_db
ID            NAME        MODE          REPLICAS      IMAGE          PORTS
nzf96q05fktm  apps_db     replicated    1/1           mysql:latest   *:3306->3306/tcp

# docker exec -it $(docker ps -f name=apps_db -q) hostname
95c15c89f891
```

Iniciamos sesión en nuestro contenedor MySQL y comprobamos nuestra base de datos creada anteriormente:

```
# docker exec -it $(docker ps -f name=apps_db -q) mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.24 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| laboratorio        |
| mydb               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.00 sec)
```