

# ADMINISTRACIÓN DE INFRAESTRUCTURAS Y SISTEMAS INFORMÁTICOS (AISI)

## Grado en Ingeniería Informática

Roberto R. Expósito ([roberto.rey.exposito@udc.es](mailto:roberto.rey.exposito@udc.es))

# DOCKER



# Contenidos

3

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# Contenidos

4

- **Introducción**
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# ¿Qué es Docker?

5

- **Docker** es una plataforma abierta para desarrollar, empaquetar y desplegar aplicaciones y servicios usando virtualización ligera basada en contenedores
  - También denominada como virtualización por compartición de *kernel* o virtualización a nivel de sistema operativo
- Actualmente, Docker goza de una gran popularidad
  - ¿Qué proporciona Docker que no proporcionan otras tecnologías de virtualización?
  - ¿Qué proporcionan los contenedores que no proporcionan otras tecnologías?



<https://www.docker.com>



# Antecedentes

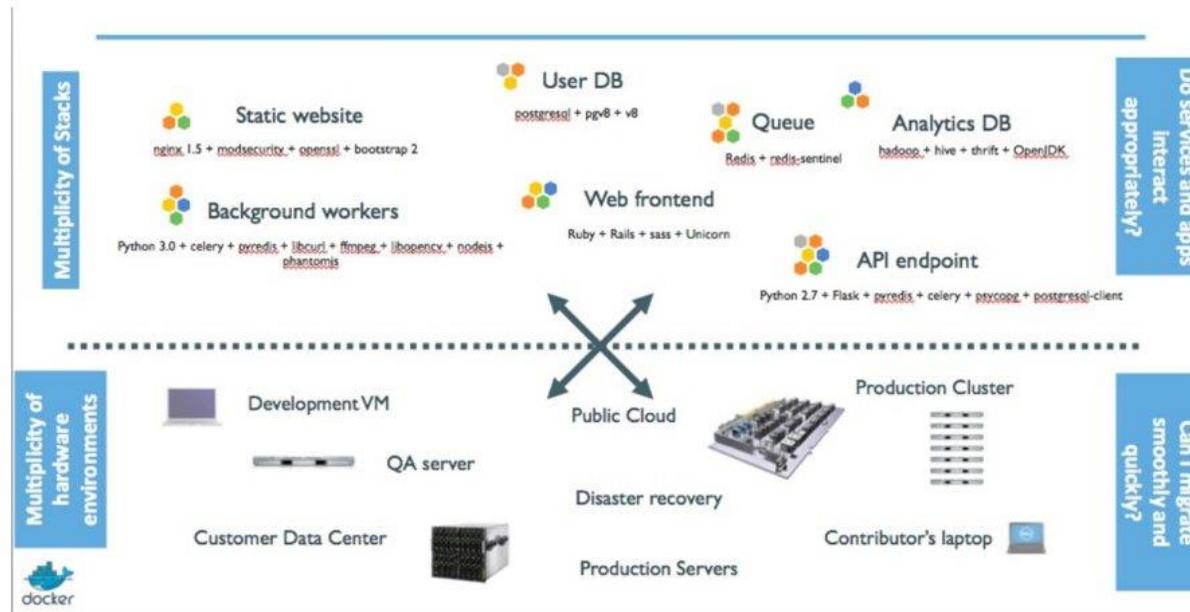
6

- Aplicaciones monolíticas: agrupan toda la funcionalidad y sus servicios en una base de código única
  - Fuertes dependencias internas
  - Ciclos de desarrollo largos
  - Entorno único
  - Difícil de escalar y migrar
- La tendencia actual se basa en desarrollar y desplegar múltiples servicios desacoplados: arquitectura de **microservicios**
  - Servicios independientes (evolución de SOA)
  - Desarrollo rápido, ágil e iterativo
  - Diferentes entornos
  - Más fácil de escalar y migrar

# Despliegue complejo

7

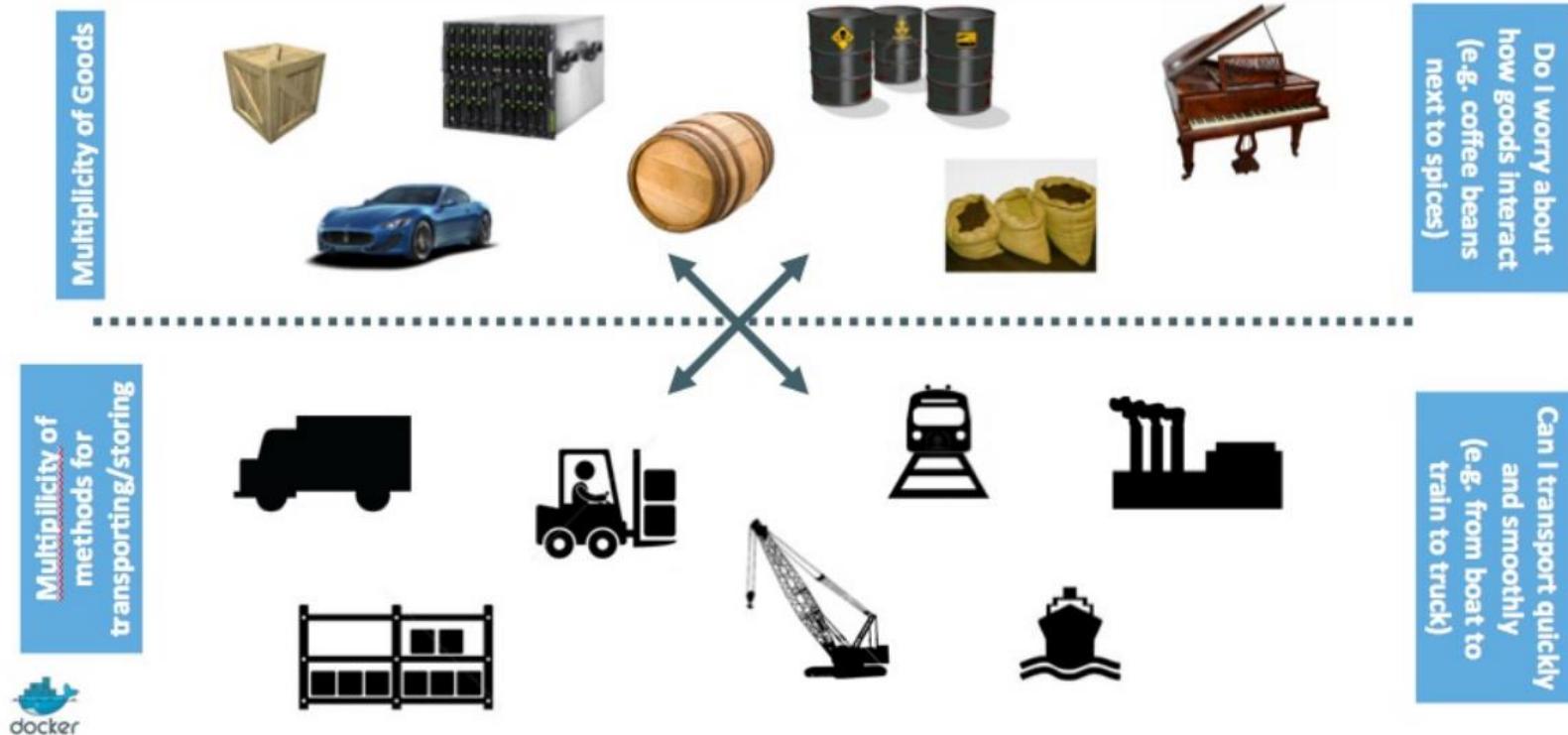
- Existencias de muchos *stacks* diferentes
  - Lenguajes, frameworks, bases de datos, ...
- A desplegar en muchos entornos distintos
  - Entornos locales de desarrollo
  - Entornos de pre-producción: *testing*, integración...
  - Entornos de producción: *on premises*, *cloud privada/pública/híbrida*...





# Analogía del transporte marítimo

8





# Analogía del transporte marítimo

9

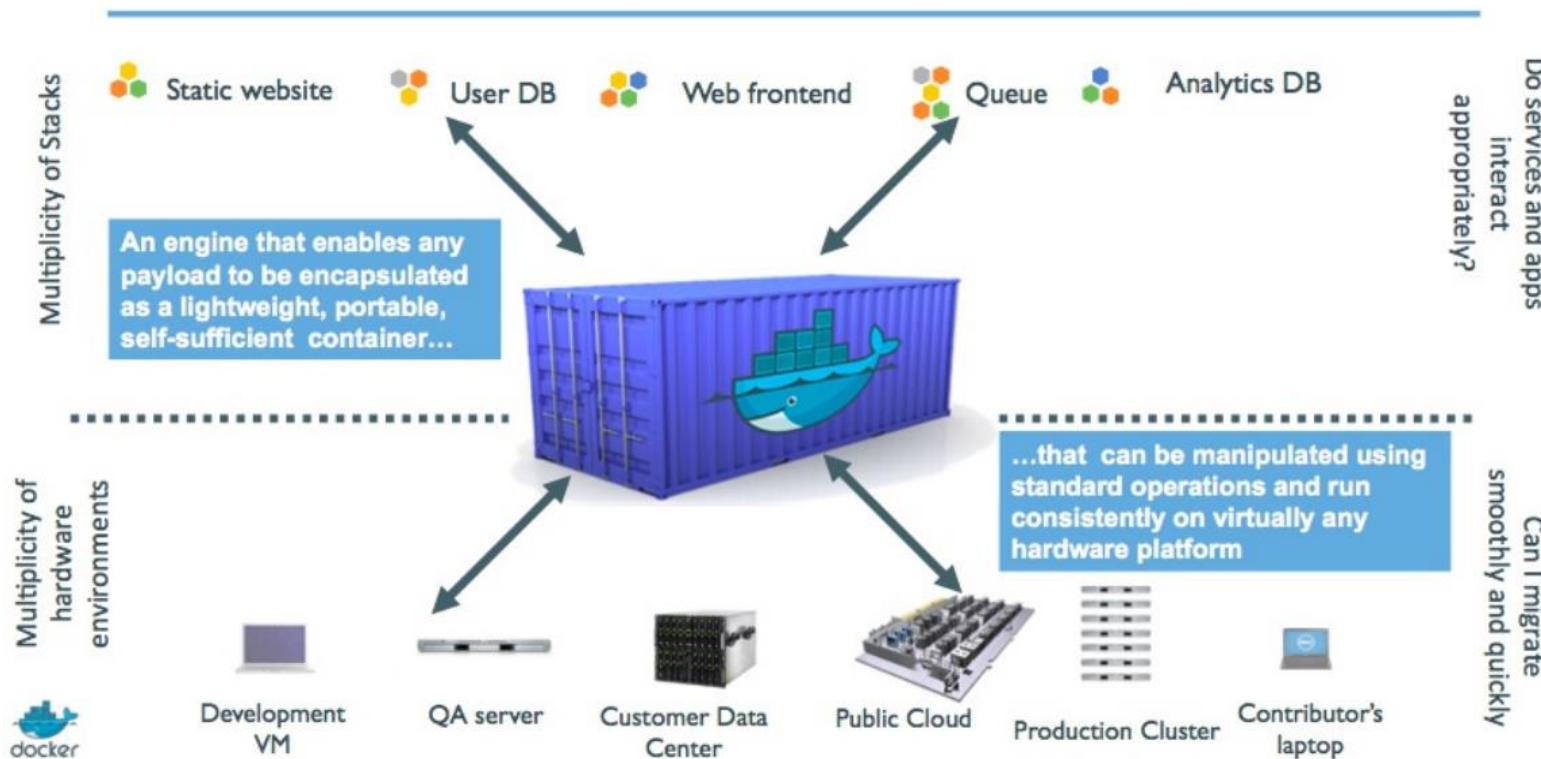
- Solución: contenedor marítimo estandarizado





# Contenedores software para aplicaciones

10

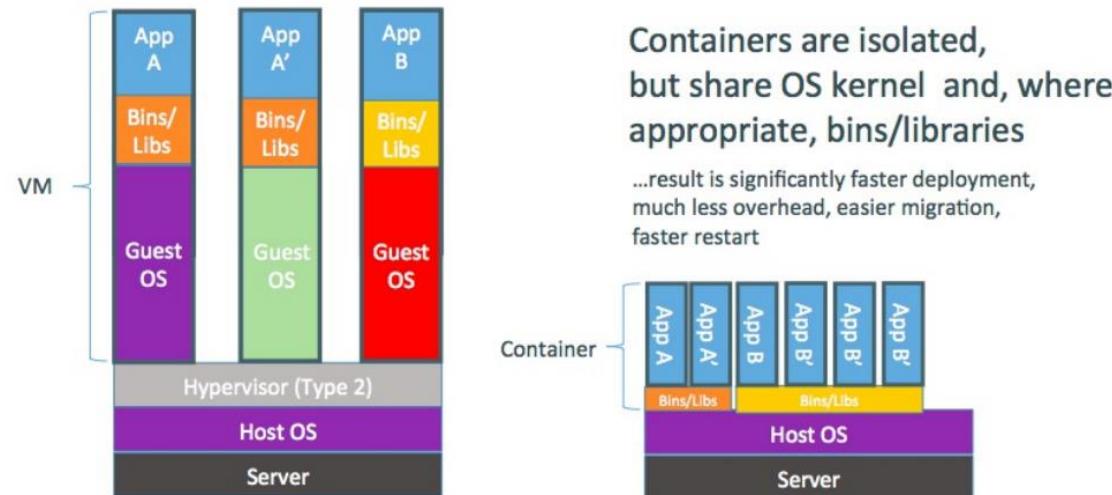




# Solución previa: Máquinas virtuales

11

- “Aseguran” entorno de desarrollo = producción
  - Configuración, versiones, dependencias, ...
- Pero introducen una serie de problemas
  - Imágenes muy grandes
  - Pérdida de rendimiento
  - Administración compleja
- Solución alternativa: contenedores software





# Contenedores

12

- Encapsulan la ejecución de un servicio/aplicación utilizando características propias que ofrece el *kernel* del sistema operativo
- Servicios aislados
  - Sistema de ficheros raíz
  - Procesos
  - Memoria
  - Redes
- No son un concepto nuevo: existen desde hace más de 15 años
  - FreeBSD Jails (1999), Linux-Vserver (2001), Solaris Containers (2004)
- Soporte por defecto en Linux desde el *kernel* 3.8
  - Versiones previas necesitaban parches específicos



# Ventajas de Docker

13

- Estandarización y simplificación de formatos
  - Definir cómo construir una imagen con *Dockerfiles*
  - Definir un *stack software* completo con Docker Compose
- Consolidación de un clúster de servidores con Docker Swarm
- Escalado, balanceo de carga, replicación...
  - Sin modificar la aplicación
- Usar los mismos contenedores para todos los entornos
  - *Testing, desarrollo, producción...*
- Creación de entornos de desarrollo en segundos/minutos

```
$ git clone ...
$ docker-compose up
```



# Contenidos

14

- Introducción
- **Conceptos básicos**
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# Mi primer contenedor

15

```
[rober@oceania ~]$ docker run ubuntu echo hello world
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
2746a4a261c9: Pull complete
4c1d20cdee96: Pull complete
0d3160e1d0de: Pull complete
c8e37668deea: Pull complete
Digest: sha256:250cc6f3fffc5cdaa9d8f4946ac79821aafb4d3afc93928f0de9336eba21aa4
Status: Downloaded newer image for ubuntu:latest
hello world
[rober@oceania ~]$ █
```

- **docker run** es el comando que permite iniciar un **contenedor**
- **ubuntu** es el nombre de una **imagen** de Docker
- Docker descarga la imagen desde un repositorio llamado **Docker Hub**
- Luego se crea el contenedor a partir de dicha imagen
- El contenedor ejecuta un binario/comando/aplicación (comando echo en este ejemplo) junto con sus parámetros (*hello world*)



# ¿Qué es una imagen?

16

- Fichero comprimido que contiene toda la información necesaria para ejecutar un contenedor
  - Todos los ficheros que necesita el contenedor (*rootfs*)
    - Ejecutables, librerías, utilidades, carpetas...
    - No es necesario instalar ni realizar ningún cambio para crear el contenedor
  - Metadatos con la configuración del contenedor
- **Un contenedor es una instancia en ejecución de una imagen**
  - De forma similar a un ejecutable (binario) y la aplicación (proceso) que se crea al ejecutarlo
  - Se pueden crear múltiples contenedores independientes a partir de la misma imagen



# ¿Qué es realmente un contenedor?

17

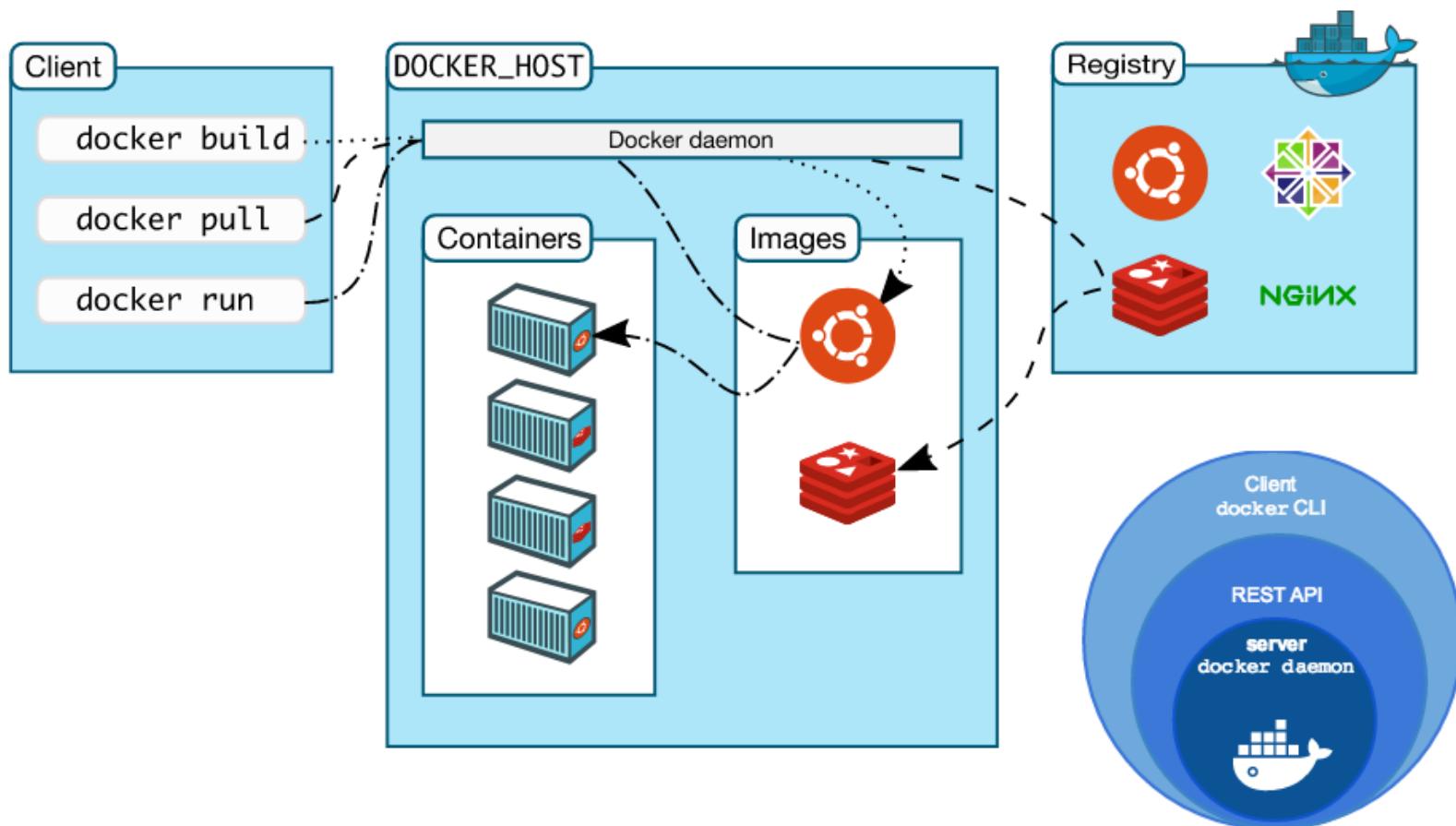
- Un contenedor no es más que un proceso en ejecución **aislado** del resto de procesos del sistema (y de otros contenedores)
  - La imagen es el sistema de ficheros completo del proceso
    - La imagen es inmutable
    - El contenedor “copia” los datos de la imagen cuando es creado y se ejecuta sobre dicha copia
  - Docker permite controlar el nivel de aislamiento de un contenedor
  - **El ciclo de vida de un contenedor está ligado al proceso que ejecuta**



# Arquitectura Docker

18

- Cliente y demonio se comunican mediante una API REST





# Creando un contenedor interactivo

19

- Ejemplo usando una imagen de Ubuntu

```
[rober@oceania ~]$ docker run -ti --name utools ubuntu:bionic bash
Unable to find image 'ubuntu:bionic' locally
bionic: Pulling from library/ubuntu
2746a4a261c9: Pull complete
4c1d20cdee96: Pull complete
0d3160e1d0de: Pull complete
c8e37668deea: Pull complete
Digest: sha256:250cc6f3f3fffc5cdaa9d8f4946ac79821aafb4d3afc93928f0de9336eba21aa4
Status: Downloaded newer image for ubuntu:bionic
root@c91f7ef811fa:/# █
```

- **-t conecta una pseudo-terminal al contenedor**
- **-i habilita modo interactivo (abre la entrada estándar del contenedor)**
- **--name <name> permite asignar un nombre al contenedor**
- **:bionic es el tag o versión de la imagen de Ubuntu**
- **bash es el binario que ejecutará el contenedor**
- **Fíjate en el prompt una vez ejecutado el contenedor**



# Creando un contenedor interactivo

20

- El contenedor funciona como una distribución Ubuntu pero compartiendo el *kernel* del equipo anfitrión (*host*)
  - Virtualización por compartición de *kernel*

```
root@c91f7ef811fa:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.3 LTS"
root@c91f7ef811fa:/# uname -a
Linux c91f7ef811fa 3.10.0-1062.9.1.el7.x86_64 #1 SMP Fri Dec 6 15:49:49 UTC 2019
x86_64 x86_64 x86_64 GNU/Linux
root@c91f7ef811fa:/# exit
exit
[rober@oceania ~]$ uname -a
Linux oceania.des.udc.es 3.10.0-1062.9.1.el7.x86_64 #1 SMP Fri Dec 6 15:49:49 UT
C 2019 x86_64 x86_64 x86_64 GNU/Linux
[rober@oceania ~]$ █
```



# Imágenes Docker

21

- De forma general, las imágenes Docker se basan en una determinada versión de un SO
- Por comodidad y simplicidad, se sirven de los repositorios de paquetes oficiales del SO
  - E.g. apt en Debian/Ubuntu, yum/dnf en RHEL/CentOS
- Por compatibilidad, mantienen la misma estructura de ficheros
  - Librerías, binarios, herramientas auxiliares...
- Aunque también se pueden ejecutar contenedores formados por un único ejecutable



# Imágenes Docker

22

- En el contenedor solo se ejecuta el proceso indicado por la línea de comandos
- Las imágenes suelen disponer de *software* mínimo
- A diferencia de una VM, no incluyen *kernel* ni *drivers* (pues no los necesitan)

```
root@2a1219c9ff50:/# ps -eaf
UID      PID  PPID  C STIME TTY          TIME CMD
root        1      0  0 10:47 pts/0      00:00:00 bash
root       12      1  0 10:48 pts/0      00:00:00 ps -eaf
root@2a1219c9ff50:/# wget
bash: wget: command not found
root@2a1219c9ff50:/# ls /boot/
root@2a1219c9ff50:/# █
```



# Ciclo de vida de un contenedor

23

- **El ciclo de vida del contenedor está ligado al proceso que ejecuta**
- Para salir del modo interactivo sin detener el contenedor, se usa la combinación ^P^Q para “demonizarlo” y que se siga ejecutando en segundo plano
  - ^P^Q = CTRL+P y CTRL+Q
- Para comprobar los contenedores **en ejecución** se puede usar el comando **docker ps**

```
root@02177a83aad0:/# [rober@oceania ~]$  
[rober@oceania ~]$ docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES  
02177a83aad0        ubuntu:bionic      "bash"            16 seconds ago   Up 16 seconds   0.0.0.0:32768->3306/tcp   utools  
[rober@oceania ~]$ █
```



# Ciclo de vida de un contenedor

24

- Es posible conectarse de nuevo al contenedor
  - `docker attach <CONTAINER>`
- Para **salir y detener** un contenedor se usa `exit`
- Sin embargo, el contenedor no se elimina
  - Sigue existiendo en disco, pero los recursos utilizados por el contenedor (p.e. CPU, memoria) son liberados
- Para mostrar **todos** los contenedores, no solos los activos (en ejecución) se usa `docker ps -a`

---

```
[rober@oceania ~]$ docker attach utools
root@02177a83aad0:/# exit
exit
[rober@oceania ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS          NAMES
[rober@oceania ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND       CREATED          STATUS           PORTS          NAMES
02177a83aad0        ubuntu:bionic      "bash"        10 minutes ago   Exited (0) 8 seconds ago
[rober@oceania ~]$ █
```



# Ciclo de vida de un contenedor

25

- Comandos útiles para controlar el ciclo de vida
  - Iniciar un contenedor detenido
    - *docker start <CONTAINER>*
  - Enviar una señal a un contenedor en ejecución
    - *docker kill -s <SIGNAL> <CONTAINER>*
    - Si no se especifica *-s*, por defecto se envía la señal KILL
  - Reiniciar un contenedor
    - *docker restart <CONTAINER>*
  - Detener un contenedor en ejecución
    - *docker stop <CONTAINER>*
    - Envía la señal SIGTERM, si en 10 segundos no se detiene, le envía SIGKILL
  - Eliminar un contenedor
    - *docker rm <CONTAINER>*
    - Indicando *-f* es posible eliminar un contenedor en ejecución (le envía la señal SIGKILL)



# Aislamiento de un contenedor

26

- Por defecto, la entrada estándar está cerrada
- El contenedor se encuentra aislado del resto de procesos del sistema y del resto de contenedores
- Tiene un sistema de ficheros propio
- Dispone de una *stack* de red propio

---

```
[rober@oceania ~]$ echo hello | tail
hello
[rober@oceania ~]$ echo hello | docker run ubuntu tail
[rober@oceania ~]$ echo hello | docker run -i ubuntu tail
hello
[rober@oceania ~]$ docker run ubuntu ps -eaf
UID      PID  PPID  C STIME TTY          TIME CMD
root      1      0  0 12:11 ?        00:00:00 ps -eaf
[rober@oceania ~]$ █
```



# Ejemplo: ejecutar el servidor web Nginx

27

- Ejecución del servidor web Nginx en segundo plano con **docker run -d**
  - Nos devuelve el **identificador del contenedor** iniciado
- Al comprobar que el servidor se está ejecutando podemos ver los puertos que el contenedor tiene abiertos
- Si intentamos acceder al puerto 80 del servidor obtenemos un error
  - El servidor web escucha en el puerto 80 del contenedor, no del host

```
[rober@oceania ~]$ docker run -d --name mynginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
8ec398bc0356: Pull complete
dfb2a46f8c2c: Pull complete
b65031b6a2a5: Pull complete
Digest: sha256:8aa7f6a9585d908a63e5e418dc5d14ae7467d2e36e1ab4f0d8f9d059a3d071ce
Status: Downloaded newer image for nginx:latest
fb65f6a26546a5197353f8e9adf85081ad943775dcc40d1f4235fe61c124598b
[rober@oceania ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
fb65f6a26546        nginx              "nginx -g 'daemon of..."   27 seconds ago    Up 26 seconds          80/tcp            mynginx
[rober@oceania ~]$ curl localhost:80
curl: (7) Failed connect to localhost:80; Conexión rehusada
[rober@oceania ~]$ █
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fb65f6a26546	nginx	"nginx -g 'daemon of..."	27 seconds ago	Up 26 seconds	80/tcp	mynginx



# Inspeccionar contenedores e imágenes

28

- Con el comando **docker inspect** podemos comprobar la configuración de contenedores e imágenes
  - Información en formato JSON
    - Con el parámetro --format se puede parsear el JSON usando una plantilla de Go
  - El contenedor hereda la configuración de la imagen
    - La configuración por defecto se puede sobreescribir con opciones del comando **docker run**

```
[rober@oceania ~]$ docker run -dti --name mynginx nginx
276b780a1ae88ea9a7178123ff0d4634ae7e68fca6f49325578917f3a42bfda0
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.IPAddress}}" mynginx
172.17.0.2
[rober@oceania ~]$ curl 172.17.0.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```



# Logs de un contenedor

29

- Docker guarda los *streams* de entrada (`stdin`) y salida (`stdout`) del proceso que se ejecuta en el contenedor
- Se pueden obtener mediante el comando `docker logs`
  - Por defecto imprime por pantalla todos los logs
  - Se pueden obtener solo los últimos logs: `--tail <nlineas>`
  - Es posible visualizar los logs en tiempo real: `--follow | -f`

```
[rober@oceania ~]$ docker logs mynginx
172.17.0.1 - - [13/Jan/2020:10:30:17 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
172.17.0.1 - - [13/Jan/2020:10:54:09 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
[rober@oceania ~]$ █
```



# Realizar cambios en un contenedor

30

- Creamos un contenedor en interactivo y modificamos la página de bienvenida de Nginx para que muestre “Hello World”
- Iniciamos otro contenedor que ejecute el servidor Nginx
- Obtenemos las IPs de ambos contenedores y comprobamos su estado con `docker ps`

```
[rober@oceania ~]$ docker run -ti --name nginx nginx bash
root@7c97d952a59e:/# echo "Hello World" > /usr/share/nginx/html/index.html
root@7c97d952a59e:/# [rober@oceania ~]$
[rober@oceania ~]$ docker run -dti --name nginx2 nginx
2992957583c226076f3e63fec7b2f60f3e7d275353246fc45752c2f4100be5a3
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.IPAddress}}" nginx
172.17.0.2
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.IPAddress}}" nginx2
172.17.0.3
[rober@oceania ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
2992957583c2        nginx              "nginx -g 'daemon of..."   About a minute ago   Up About a minute   80/tcp                nginx2
7c97d952a59e        nginx              "bash"                  2 minutes ago       Up 2 minutes        80/tcp                nginx
[rober@oceania ~]$ █
```



# Realizar cambios en un contenedor

31

- Accedemos con *curl* a ambos contenedores
  - El primer contenedor no está ejecutando el servidor Nginx
  - En el segundo contenedor, vemos la página de bienvenida por defecto de Nginx
    - Una vez creado, la configuración de un contenedor es immutable
    - Excepto la configuración de red y algún otro dato como su nombre

```
[rober@oceania ~]$ curl 172.17.0.2
curl: (7) Failed connect to 172.17.0.2:80; Conexión rehusada
[rober@oceania ~]$ curl 172.17.0.3
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
```



# Realizar cambios en un contenedor

32

- Comprobamos el comando que ejecutan ambos contenedores
- Ejecutamos el mismo comando en el contenedor interactivo para iniciar el servidor Nginx y accedemos al mismo: obtenemos "Hello World"

```
[rober@oceania ~]$ docker ps --no-trunc
CONTAINER ID        IMAGE       COMMAND
NAMES
2992957583c226076f3e63fec7b2f60f3e7d275353246fc45752c2f4100be5a3   nginx      "nginx -g 'daemon off;'"    22 minutes ago   Up 22 minutes   80/tcp
nginx2
7c97d952a59eddaf12838d70de0d466f70928e821a37631ba5035238cf8e1141   nginx      "bash"                  23 minutes ago   Up 23 minutes   80/tcp
nginx
```

```
[rober@oceania ~]$ docker attach 7c97d952a59e
root@7c97d952a59e:/# nginx -g 'daemon off;'
```

```
172.17.0.1 - - [13/Jan/2020:11:40:58 +0000] "GET / HTTP/1.1" 200 12 "-" "curl/7.29.0" "-"
```

```
rober@oceania:~
```

```
Archivo Editar Ver Buscar Terminal Ayuda
[rober@oceania ~]$ curl 172.17.0.2
Hello World
[rober@oceania ~]$
```

- Para cambiar la configuración de un contenedor manteniendo los cambios realizados tenemos que:
  - Crear una imagen a partir del contenedor actual
  - Crear un nuevo contenedor a partir de la nueva imagen



# Imágenes y contenedores (recordatorio)

33

- Imagen

- Es como una "plantilla" que permite ejecutar contenedores
- Compuesta por un conjunto de ficheros y metadatos
  - Ejemplo de metadatos: comando a ejecutar, puertos públicos...
- Es de sólo lectura
- Se descargan (por defecto) de Docker Hub

- Contenedor

- Proceso o conjunto de procesos encapsulados y aislados del resto del sistema
- Se ejecutan en una copia de la imagen
- Con el comando `docker run` se puede ejecutar un nuevo contenedor a partir de una imagen
- Su ciclo de vida está ligado al proceso que ejecuta



# Contenidos

34

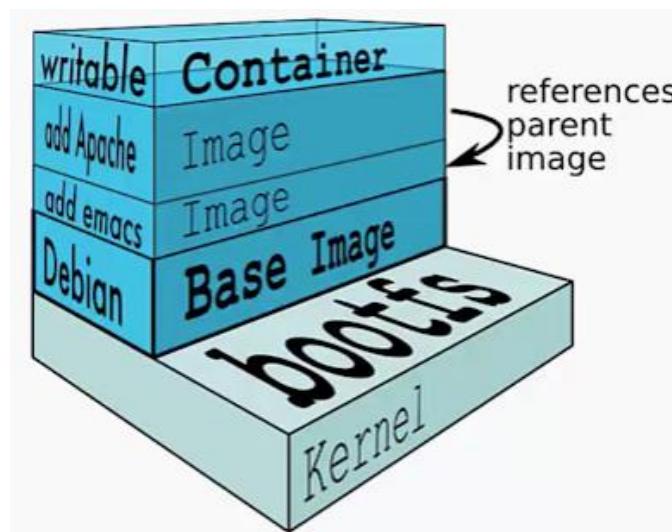
- Introducción
- Conceptos básicos
- **Imágenes**
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# ¿Qué es una imagen?

35

- Conjunto de ficheros que forman el *rootfs* del contenedor
- Son de sólo lectura
- Formada por capas superpuestas por niveles

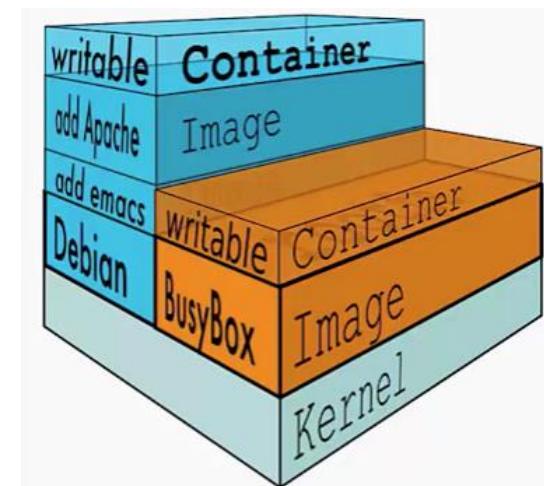
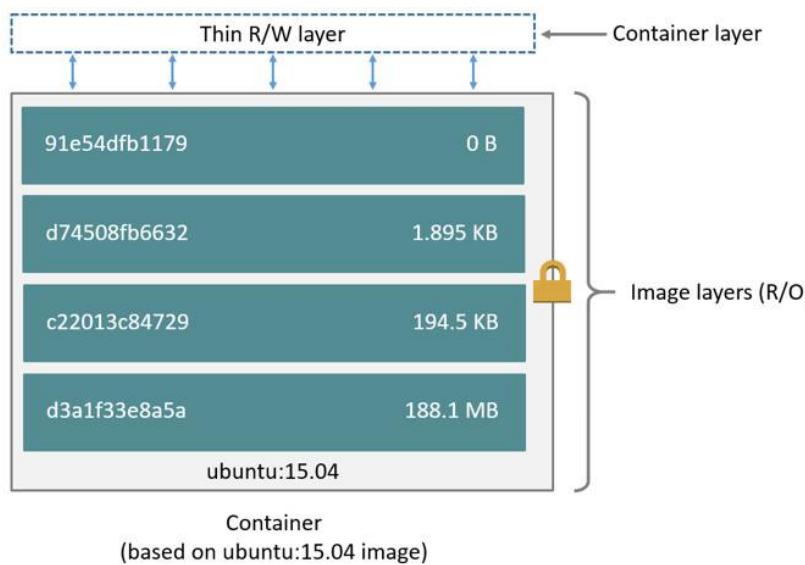




# Capas de una imagen

36

- Cada capa añade, modifica o elimina ficheros
  - Las imágenes comparten capas para optimizar disco, memoria y transferencia de datos
    - Estrategia Copy-on-Write (CoW) como optimización
  - El contenedor se ejecuta sobre un sistema de ficheros que es una nueva capa de lectura/escritura creada sobre la capa de sólo lectura de la imagen base





# Creación y modificación de imágenes

37

- A partir de un fichero tar
  - **docker import**: imagen sin metadatos
  - **docker load**: imagen con metadatos
- A partir de un contenedor en ejecución: **docker commit**
  - Hace de sólo lectura la capa de lectura/escritura del contenedor
  - Guarda todos los cambios hechos por el contenedor
  - La nueva imagen está formada por la nueva capa y las capas de la imagen base
- A partir de un fichero *Dockerfile*: **docker build**
  - Proceso repetible y extensible de creación de contenedores
  - Forma estándar de creación de imágenes
  - Paradigma IaC



# Creación y modificación de imágenes

38

- Creamos un contenedor que modifica la página de bienvenida del servidor web Nginx para que muestre “Hello World”
- Comprobamos los cambios realizados por el contenedor sobre la imagen base con el comando **docker diff**
- Creamos la nueva imagen llamada **nginx:hello** mediante el comando **docker commit**
  - Este comando crea una *snapshot* del estado actual del contenedor
  - La nueva imagen se puede listar usando **docker image ls**

```
[rober@oceania ~]$ docker run --name hello-nginx nginx bash -c "echo Hello World > /usr/share/nginx/html/index.html"
[rober@oceania ~]$ docker diff hello-nginx
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
[rober@oceania ~]$ docker commit hello-nginx nginx:hello
sha256:05c4a40d879179f34da099be2b0631d617d4f7cf5dec8b08f77fa17c8a5af3c8
[rober@oceania ~]$ █
```



# Creación y modificación de imágenes

39

- Creamos un nuevo contenedor desde la imagen creada (`nginx:hello`) y obtenemos su dirección IP
  - ¿Problema? No se está ejecutando el servidor Nginx
  - La nueva imagen hereda la configuración del contenedor, incluyendo el comando por defecto que se ejecuta
- Luego creamos un nuevo contenedor con el comando necesario para poder ejecutar el servidor Nginx

```
[rober@oceania ~]$ docker run -d --name mynginx nginx:hello  
1f2832a0a0415336c3053f9d39b823764a746bd6811a8af220c49f9a5ef1eb21  
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.IPAddress}}" mynginx
```

COLUMN	COMMAND	CREATED	STATUS	PORTS	NAMES
CONTAINER ID	IMAGE	"bash -c 'echo Hello...'"	6 seconds ago	Exited (0) 6 seconds ago	mynginx
1f2832a0a041	nginx:hello				
[rober@oceania ~]\$ docker rm mynginx mynginx					
[rober@oceania ~]\$ docker run -d --name mynginx nginx:hello nginx -g 'daemon off;' 727d6158453111d181c7f978cba390a9ba0ca08ba6b66ee50aba4e6d20be8990					
[rober@oceania ~]\$ docker inspect --format "{{.NetworkSettings.IPAddress}}" mynginx 172.17.0.2					
[rober@oceania ~]\$ curl 172.17.0.2 Hello World					
[rober@oceania ~]\$ █					



# Creación de imágenes interactiva

40

- La aproximación seguida en el ejemplo previo para crear una imagen a partir de una sesión interactiva en la que se realizan todos los cambios necesarios presenta varios problemas:
  - Es un proceso no automático y farragoso de replicar
  - Cierta información como las variables de entorno se pierden
  - Es fácil incluir cambios innecesarios en la imagen
  - Imágenes opacas
- Mejor solución siguiendo el paradigma IaC
  - Uso de **Dockerfiles** y comando **docker build**
  - El siguiente apartado del tutorial se centrará en este aspecto



# Eliminación de imágenes

41

- Eliminar una imagen: **docker rmi <image>**
  - La opción **-f** permite forzar la eliminación si hay algún contenedor que todavía referencia la imagen

```
[rober@oceania ~]$ docker run -d --name mynginx nginx:hello2
e1c9763103a6dd7608cfcc6e8d01ab3bf2145a0e63f3132093f02e1947cdd69f
[rober@oceania ~]$ docker stop e1c9763103
e1c9763103
[rober@oceania ~]$ docker rmi nginx:hello2
Error response from daemon: conflict: unable to remove repository reference "nginx:hello2" (must force) - container e1c9763103a6 is using its referenced image 167137dd7030
[rober@oceania ~]$ docker rmi -f nginx:hello2
Untagged: nginx:hello2
Deleted: sha256:167137dd7030655c4115acf3f7bb9e2eebd512c53d948ee928545c59ef03c7da
[rober@oceania ~]$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
e1c9763103a6        167137dd7030        "nginx -g 'daemon of..."   About a minute ago   Exited (0)   About a minute ago
[rober@oceania ~]$ docker rm e1c9763103a6
e1c9763103a6
[rober@oceania ~]$ █
```



# Repositorios de imágenes

42

- Cuando creamos una imagen como en los ejemplos anteriores, esta se almacena en local en una caché de imágenes
  - El comando **docker images** permite listarlas
- Para distribuir una imagen creada en local a otra máquina distinta:
  - Empaquetarla en un fichero con **docker save** y transferirla (por la red, pendrive, etc)
  - Almacenarla en un registro de imágenes y obtenerla desde ahí
    - Comandos: **docker push** y **docker pull**

```
[rober@oceania ~]$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	hello	05c4a40d8791	40 minutes ago	126MB
nginx	latest	c7460dfcab50	3 days ago	126MB
busybox	latest	6d5fcfe5ff17	2 weeks ago	1.22MB
ubuntu	bionic	549b9b86cb8d	3 weeks ago	64.2MB
ubuntu	latest	549b9b86cb8d	3 weeks ago	64.2MB
bash	latest	723771701d55	2 months ago	15.2MB



# Registros Docker

43

- Un registro Docker es un repositorio de imágenes
  - El registro asignado a una imagen está definido en su nombre
- El registro público se llama **Docker Hub**
  - Las imágenes públicas son gratuitas
  - Solo se paga por acceso a un repositorio privado
- Es posible instalar y gestionar registros propios
  - Docker Trusted Registry (de pago)
    - <https://docs.docker.com/ee/dtr>
  - Docker Registry (open source)
    - <https://docs.docker.com/registry>



# Guardar una imagen en Docker Hub

44

- Es posible subir una imagen a un registro
  - ***docker push <name[:TAG]>***
- Intentamos subir a Docker Hub nuestra nueva imagen `nginx:hello`
  - Error! No tenemos permisos para subir una imagen
  - Sí que podríamos subir la imagen a un registro privado
    - Usaríamos ***docker pull*** para obtenerla/descargarla
    - ***docker run*** hace un *pull* automático cuando la imagen no está disponible en local

```
[rober@oceania ~]$ docker push nginx:hello
The push refers to repository [docker.io/library/nginx]
660f524ad658: Preparing
c26e88311e71: Preparing
17fde96446df: Preparing
556c5fb0d91b: Preparing
denied: requested access to the resource is denied
[rober@oceania ~]$ █
```



# Contenidos

45

- Introducción
- Conceptos básicos
- Imágenes
- **Dockerfiles**
- Redes de contenedores
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# ¿Qué es un *Dockerfile*?

46

- Es un fichero que contiene todos los pasos necesarios o instrucciones para crear una imagen Docker
- Proporciona un proceso **automático** y **replicable** para la creación de imágenes
- Los pasos del *Dockerfile* se ejecutan **secuencialmente**
  - Cada paso crea una imagen válida intermedia desde la que parte el paso siguiente
  - Las imágenes intermedias se guardan en una caché interna
  - Es posible iniciar un contenedor a partir de ellas
- Para generar una imagen a partir de un fichero *Dockerfile* se usa el comando ***docker build***



# Instrucciones básicas

47

- **FROM:** indica la imagen base desde la que se parte
  - Primera instrucción (exceptuando #comentarios iniciales)
    - echo "FROM redis" >> Dockerfile
- **RUN:** ejecuta el comando indicado
  - Todos los comandos deben ser **no interactivos**
    - echo "RUN apt-get install -y unzip" >> Dockerfile
  - Puede haber múltiples instrucciones RUN
  - Cada instrucción RUN crea un nuevo contenedor a partir de la imagen previa
  - Una instrucción RUN sólo modifica ficheros de la imagen



# Creando una imagen

48

- Comando **docker build**

- Se le pasa como parámetro el directorio donde está el *Dockerfile*
- El parámetro **-t** permite indicar el nombre y tag de la imagen siguiendo el formato: **nombre[:tag]**

```
[rober@oceania ~]$ docker build -t myredis redis
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM redis
--> 9b188f5fb1e6
Step 2/3 : RUN apt-get update
--> Running in 2266e72aae26
Get:1 http://deb.debian.org/debian buster InRelease [122 kB]
Get:3 http://deb.debian.org/debian buster-updates InRelease [49.3 kB]
Get:4 http://deb.debian.org/debian buster/main amd64 Packages [7908 kB]
Get:2 http://security-cdn.debian.org/debian-security buster/updates InRelease [65.4 kB]
Get:5 http://security-cdn.debian.org/debian-security buster/updates/main amd64 Packages [171 kB]
Get:6 http://deb.debian.org/debian buster-updates/main amd64 Packages [5792 B]
Fetched 8321 kB in 2s (3745 kB/s)
Reading package lists...
Removing intermediate container 2266e72aae26
--> fc5d99b4a7b6
```



# Creando una imagen

49

- La imagen final del paso 1 es **9b188f5fb1e6**, que a su vez es la imagen base del siguiente paso
- Luego se crea un contenedor intermedio (Running in **2266e72aae26**) sobre el que se ejecutará el paso 2
  - Sobre ese contenedor intermedio se ejecuta el comando definido por RUN
    - En este caso: *apt-get update*
  - Si la ejecución es satisfactoria, se hace *commit* del contenedor y se elimina

```
[rober@oceania ~]$ docker build -t myredis redis
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM redis
--> 9b188f5fb1e6
Step 2/3 : RUN apt-get update
--> Running in 2266e72aae26
Get:1 http://deb.debian.org/debian buster InRelease [122 kB]
Get:3 http://deb.debian.org/debian buster-updates InRelease [49.3 kB]
Get:4 http://deb.debian.org/debian buster/main amd64 Packages [7908 kB]
Get:2 http://security-cdn.debian.org/debian-security buster/updates InRelease [65.4 kB]
Get:5 http://security-cdn.debian.org/debian-security buster/updates/main amd64 Packages [171 kB]
Get:6 http://deb.debian.org/debian buster-updates/main amd64 Packages [5792 B]
Fetched 8321 kB in 2s (3745 kB/s)
Reading package lists...
Removing intermediate container 2266e72aae26
--> fc5d99b4a7b6
```



# Creando una imagen

50

- El paso 3 es el último y devuelve la imagen final (**8d0314c7b342**)

```
Step 3/3 : RUN apt-get install -y unzip
--> Running in lddb510f9e78
Reading package lists...
Building dependency tree...
Reading state information...
The following package was automatically installed and is no longer required:
  lsb-base
Use 'apt autoremove' to remove it.
Suggested packages:
  zip
The following NEW packages will be installed:
  unzip
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 172 kB of archives.
After this operation, 580 kB of additional disk space will be used.
Get:1 http://deb.debian.org/debian buster/main amd64 unzip amd64 6.0-23+deb10u1 [172 kB]
debconf: delaying package configuration, since apt-utils is not installed
Fetched 172 kB in 0s (663 kB/s)
Selecting previously unselected package unzip.
(Reading database ... 6466 files and directories currently installed.)
Preparing to unpack .../unzip_6.0-23+deb10u1_amd64.deb ...
Unpacking unzip (6.0-23+deb10u1) ...
Setting up unzip (6.0-23+deb10u1) ...
Removing intermediate container lddb510f9e78
--> 8d0314c7b342
Successfully built 8d0314c7b342
Successfully tagged _myredis:latest
```



# Caché

51

- Docker mantiene una cache de los diferentes pasos que se ejecutan con sus respectivas imágenes intermedias
  - Antes de cada paso, se comprueba si la misma secuencia fue ejecutada previamente
  - La caché solo analiza las líneas del *Dockerfile*
  - Si se vuelve a crear la imagen del ejemplo previo, la instrucción “RUN apt-get update” no se ejecutaría aunque el repositorio se haya actualizado
    - Ver figura (**Using cache**)
  - Se puede ignorar la caché con la opción **--no-cache**

```
[rober@oceania ~]$ docker build -t myredis redis
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM redis
    --> 9b188f5fb1e6
Step 2/3 : RUN apt-get update
    --> Using cache
    --> fc5d99b4a7b6
Step 3/3 : RUN apt-get install -y unzip
    --> Using cache
    --> 8d0314c7b342
Successfully built 8d0314c7b342
Successfully tagged myredis:latest
[rober@oceania ~]$ █
```



# Historial de una imagen

52

- El comando **docker history** muestra todas las capas de una imagen
  - Cada capa se corresponde con una instrucción del *Dockerfile*
  - Los parámetros de la instrucción RUN se pasan al comando "/bin/sh -c"
  - La instrucción FROM se sustituye por las instrucciones de la imagen base usada (9b188f5fb1e6, en la figura)
  - Las tres capas creadas durante la construcción de nuestra imagen **myredis** aparecen con sus correspondientes imágenes intermedias (tienen identificadores)
  - Capas sin imagen asociada (*missing*) pertenecen a la imagen base, ya que al repositorio Docker Hub no se suben las imágenes intermedias
  - La columna "SIZE" muestra el tamaño de cada capa, algunas no afectan al tamaño final (instrucciones de configuración)

---

[rober@oceania ~]\$ docker history myredis	IMAGE	CREATED	CREATED BY	SIZE
	492873ed8018	52 seconds ago	/bin/sh -c apt-get install -y unzip	1.29MB
	9e4bcc5a789c	53 seconds ago	/bin/sh -c apt-get update	17.4MB
	9b188f5fb1e6	11 days ago	/bin/sh -c #(nop) CMD ["redis-server"]	0B
	<missing>	11 days ago	/bin/sh -c #(nop) EXPOSE 6379	0B
	<missing>	11 days ago	/bin/sh -c #(nop) ENTRYPOINT ["docker-entry...	0B
	<missing>	11 days ago	/bin/sh -c #(nop) COPY file:df205a0ef6e6df89...	374B
	<missing>	11 days ago	/bin/sh -c #(nop) WORKDIR /data	0B



# Instrucción WORKDIR

53

- La instrucción **WORKDIR** permite establecer el **directorío de trabajo** para otras instrucciones (RUN, CMD, ENTRYPOINT, COPY, ADD)
  - La instrucción “RUN cd /opt/redis” solo tiene efecto dentro de dicha instrucción, ya que al no modificar ningún fichero no queda reflejada en la imagen resultado
  - Otra instrucción RUN que se ejecute a continuación de la anterior partirá del directorio de trabajo por defecto (el directorio raíz /)
    - “RUN cp file /data” asume que el fichero “file” existe en / (y no en /opt/redis)
  - Solución: anidar comandos en una única instrucción RUN con &&
    - RUN cd /opt/redis && cp file /data
    - Se puede dividir los comandos en diferentes líneas con \
    - Si un comando falla, los comandos a continuación no se ejecutan
    - **Anidar comandos reduce el número de capas de la imagen**
  - Como alternativa, también se puede cambiar el directorio de trabajo con WORKDIR
    - Sucesivas instrucciones WORKDIR se sobrescriben entre sí
    - También sobrescribe el WORKDIR de la imagen base/padre



# Instrucción ENV

54

- Una variable de entorno que se defina en una instrucción RUN solo está disponible de forma local para dicha instrucción
  - `RUN export REDIS_HOME=/opt/redis && cp $REDIS_HOME/file /data`
- Si la imagen necesitase el valor de la variable de entorno para funcionar correctamente, es necesario definirla con una instrucción ENV
  - También se pueden anidar instrucciones ENV, aunque si existen dependencias entre ellas deben definirse en líneas separadas
  - Cada instrucción ENV también crea una nueva capa
- También es posible definir variables de entorno en tiempo de ejecución usando el parámetro -e VAR=VALUE del comando docker run



# Instrucción COPY

55

- La instrucción **COPY** permite copiar ficheros locales almacenados en el *host* al sistema de ficheros de la imagen
- Los ficheros a copiar deben estar dentro del directorio donde se encuentra el *Dockerfile*
  - Dicho directorio se conoce como el “*build context*”
  - También es posible copiar directorios
- Esta instrucción crea un nuevo contenedor intermedio así como una nueva capa de la imagen
- Los permisos se mantienen, pero el propietario y el grupo se cambian a UID y GID 0 (*root*)



# Instrucción COPY

56

- Ejemplo

- Creamos una imagen “personalizada” de Ubuntu que incluye un nuevo fichero llamado **file**
- Directorio “*build context*” con los ficheros *Dockerfile* y *file*

---

```
[rober@oceania ~]$ ls -l ubuntu/
total 8
-rw-rw-r-- 1 rober rober 53 ene 14 12:40 Dockerfile
-rw-rw-r-- 1 rober rober 12 ene 14 12:40 file
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
RUN apt-get update
COPY file /opt
[rober@oceania ~]$ cat ubuntu/file
Hello World
[rober@oceania ~]$ □
```



# Instrucción COPY

57

- Ejemplo
  - Creación de la imagen con *docker build*

```
[rober@oceania ~]$ docker build -t myubuntu ubuntu/
Sending build context to Docker daemon 3.072kB
Step 1/3 : FROM ubuntu:bionic
--> 549b9b86cb8d
Step 2/3 : RUN apt-get update
--> Running in 77d13e6a362b
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
Get:2 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Get:3 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:4 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
          Fetched 17.4 MB in 5s (3768 kB/s)
          Reading package lists...
          Removing intermediate container 77d13e6a362b
          --> d39fdb6bde1b
Step 3/3 : COPY file /opt
--> 4a4278cb19ba
Successfully built 4a4278cb19ba
Successfully tagged _myubuntu:latest
```



# Instrucción COPY

58

- **Ejemplo**

- **Creamos un nuevo contenedor usando la nueva imagen y mostramos el contenido del fichero**
  - Fíjate en el propietario y grupo del fichero dentro del contenedor

```
[rober@oceania ~]$ docker run -ti myubuntu
root@4dffed5b6bd6:/# cat /opt/file
Hello World
root@4dffed5b6bd6:/# ls -l /opt/file
-rw-rw-r-- 1 root root 12 Jan 14 11:40 /opt/file
root@4dffed5b6bd6:/# █
```



# Instrucción CMD

59

- La instrucción **CMD** define el **comando por defecto** que ejecuta un contenedor cuando **no** se especifica nada en `docker run`
  - Sucesivos CMD se sobrescriben entre sí (también el de la imagen base)

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
RUN apt-get update
COPY file /opt
CMD cat /opt/file
[rober@oceania ~]$ docker build -t myubuntu ubuntu/
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM ubuntu:bionic
--> 549b9b86cb8d
Step 2/4 : RUN apt-get update
--> Using cache
--> d39fdb6bde1b
Step 3/4 : COPY file /opt
--> Using cache
--> 4a4278cb19ba
Step 4/4 : CMD cat /opt/file
--> Running in 12c63c141bb4
Removing intermediate container 12c63c141bb4
--> 7439da41a645
Successfully built 7439da41a645
Successfully tagged myubuntu:latest
[rober@oceania ~]$ docker run -ti myubuntu
Hello World
[rober@oceania ~]$ █
```



# Instrucción ENTRYPPOINT

60

- La instrucción **ENTRYPOINT** define un **comando base** al que se le pasan como parámetros los argumentos indicados con *docker run*
  - Si no se especifica ENTRYPOINT en el *Dockerfile*, por defecto es: **/bin/sh -c**
  - Si no se especifica ningún argumento en *docker run*, se pasa CMD como parámetro por defecto al ENTRYPOINT
    - Se ejecutaría: **/bin/sh -c CMD**
- Una instrucción ENTRYPOINT sobrescribe el de la imagen base
  - Pero se puede invocar su ENTRYPOINT previamente (si es necesario)
  - Con *docker history* podemos ver el ENTRYPOINT de una imagen
- Se puede sobrescribir en *docker run* con la opción **--entrypoint**
- Ejemplo de uso (sin mostrar el proceso de creación de la imagen)

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENV name Rober
ENTRYPOINT echo "Hello, $name"
[rober@oceania ~]$ docker run -ti myubuntu
Hello, Rober
[rober@oceania ~]$ █
```



# Argumentos de RUN, CMD, ENTRYPOINT

61

- Se pueden especificar de dos maneras
  - En formato **shell**: <instruction> <command>
    - El comando se ejecuta en una shell: /bin/sh -c <command>
    - Ejemplos:
      - RUN apt-get install python3
        - Se ejecutaría: /bin/sh -c "apt-get install python3"
        - CMD echo "Hello world"
        - ENTRYPOINT echo "Hello world"
  - En formato **exec**: <instruction> ["executable", "param1", ...]
    - El executable se ejecuta directamente (sin una shell)
    - Ejemplos:
      - RUN ["apt-get", "install", "python3"]
      - CMD ["/bin/echo", "Hello world"]
      - ENTRYPOINT ["/bin/echo", "Hello world"]



# Argumentos de RUN, CMD, ENTRYPOINT

62

- Se pueden especificar de dos maneras
  - En formato **shell**: <instruction> <command>
  - En formato **exec**: <instruction> ["executable", "param1", ...]

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENV name Rober
ENTRYPOINT echo "Hello, $name"
[rober@oceania ~]$ docker run -ti myubuntu
Hello, Rober
[rober@oceania ~]$ █
```

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENV name Rober
ENTRYPOINT ["/bin/echo", "Hello, $name"]
[rober@oceania ~]$ docker run -ti myubuntu
Hello, $name
[rober@oceania ~]$ █
```

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENV name Rober
ENTRYPOINT ["/bin/bash", "-c", "echo Hello, $name"]
[rober@oceania ~]$ docker run -ti myubuntu
Hello, Rober
[rober@oceania ~]$ █
```



# Argumentos de RUN, CMD, ENTRYPOINT

63

- ENTRYPOINT ignora el CMD y cualquier argumento que se le pase a docker run cuando se usa el formato shell

---

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["World"]
[rober@oceania ~]$ docker run -ti myubuntu
Hello World
[rober@oceania ~]$ docker run -ti myubuntu Rober
Hello Rober
[rober@oceania ~]$ █
```

---

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
ENTRYPOINT echo Hello
CMD ["World"]
[rober@oceania ~]$ docker run -ti myubuntu
Hello
[rober@oceania ~]$ docker run -ti myubuntu Rober
Hello
[rober@oceania ~]$ █
```



# Proceso inicial de un contenedor

64

- Docker está orientado a la ejecución de un proceso por contenedor
- El proceso con el que se crea el contenedor tiene PID=1, siendo este proceso:
  - El comando definido con ENTRYPOINT
  - O el comando especificado en *docker run*
  - O el comando definido con CMD
- Este proceso tiene un rol especial
  - Define el ciclo de vida del contenedor
  - Recibe las señales enviadas con *docker stop* y *docker kill*
  - Su salida estándar y de error es capturada por el demonio Docker



# Instrucción EXPOSE

65

- La instrucción **EXPOSE** permite especificar en qué puerto(s) un contenedor escuchará peticiones **pero sin publicarlos/mapearlos** a puertos del host
  - Ejemplo: EXPOSE 80/tcp 443/tcp 80/udp
  - Se considera un tipo de “documentación” entre la persona que crea el *Dockerfile* y/o genera la imagen y la persona que ejecuta el contenedor
- La opción -P de *docker run* publica (mapea) todos los puertos declarados con EXPOSE a puertos aleatorios del host
- La opción -p <hostPort:containerPort> de *docker run* publica el puerto del contenedor especificado a un puerto concreto del host
- El comando **docker ps** muestra los puertos y *mappings* existentes

```
[rober@oceania ~]$ docker run -d -P redis
61e7017c8d1ddfb8fcabc03db2c3df934d2141b910adac31f681c436903a3f92
[rober@oceania ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                         NAMES
61e7017c8d1d        redis              "docker-entrypoint.s..."   5 seconds ago       Up 3 seconds          0.0.0.0:32768->6379/tcp   goofy_habit
[rober@oceania ~]$ █
```



# Instrucción HEALTHCHECK

66

- Permite especificar un comando para detectar si el contenedor está "vivo" (en ejecución)
  - `HEALTHCHECK [OPTIONS] CMD <command>`
  - `HEALTHCHECK NONE`
    - Deshabilita cualquier *healthcheck* heredado de la imagen base
- El estado del contenedor se deriva del resultado del comando
  - 0: *success* -> el contenedor está vivo
  - 1: *unhealthy* -> el contenedor no está funcionando correctamente
- Ejemplo: comprobando si un servidor web responde
  - `HEALTHCHECK CMD curl --fail http://127.0.0.1:80 || exit 1`
- Opciones
  - `--interval=DURATION` (30 segundos por defecto)
  - `--timeout=DURATION` (30 segundos por defecto)
  - `--retries=N` (3 por defecto)



# Instrucción ADD

67

- La instrucción **ADD** extiende la funcionalidad de COPY ya que también permite obtener ficheros remotos desde una URL
  - Los ficheros siempre se descargan antes de comprobar si han cambiado
  - Los ficheros descargados tienen permisos 600 y UID/GID 0
- Ejemplo:
  - ADD <https://myapp.com/app.jar> /opt/myapp
- ADD también permite descomprimir archivos gzip, xz y tar locales, pero no ficheros remotos
  - ADD fichero.zip /opt/myapp



# Instrucción USER

68

- La instrucción **USER** cambia el identificador de usuario (UID) y/o grupo (GID) que ejecuta la imagen
  - `USER <user>[:<group>]`
  - `USER <UID>[:<GID>]`
- Ejemplos:
  - `USER rober`
  - `USER 1000`
  - `USER 1000:nogroup`
  - `USER 1000:1010`
- El usuario o grupo especificados **deben existir**
- Recomendable su uso desde el punto de vista de la seguridad
- Las instrucciones RUN se ejecutan por defecto como *root* (UID 0)
- Es posible cambiar múltiples veces de usuario en un *Dockerfile*



# Buenas prácticas (I)

69

- Aprovechar la caché de Docker
  - El orden de las instrucciones es relevante
  - Comandos permanentes y no variables mejor al principio del *Dockerfile*
  - Copia de ficheros que se modifican con frecuencia o configuraciones cambiantes mejor hacia el final
- Minimizar el número de capas
  - Tener en cuenta que casi todas las instrucciones crean una nueva capa
  - Las instrucciones pueden ocupar varias líneas usando \
  - Encontrar balance entre legibilidad y número de instrucciones
- Instalar el mínimo software necesario
  - Reduce la complejidad, tamaño de la imagen...
- Eliminar todo lo que no sea necesario
  - Código fuente una vez que ha sido compilado
  - Archivos comprimidos tras ser descomprimidos
  - Dependencias innecesarias, caché de los gestores de paquetes, ...



## Buenas prácticas (II)

70

- Cuando sea posible, partir de una imagen oficial como base
- Usar imágenes base lo más ligeras posible
- Es preferible definir las instrucciones CMD y ENTRYPOINT con formato exec
- Usar siempre puertos por defecto en EXPOSE
- Usar USER siempre que sea posible
- Usar siempre versiones concretas de imágenes y software



# Contenidos

71

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- **Redes**
- Almacenamiento
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# Publicando puertos de un contenedor

72

- Por defecto, un contenedor solo puede ser accedido desde redes locales
- Como hemos visto, la opción **-P** de `docker run` **publica (mapea)** todos los puertos **expuestos** por el contenedor
  - Los mapea a puertos aleatorios del host
- En este ejemplo el puerto 80 (TCP) del contenedor se publica/mapea al puerto 32770 del host

---

```
[rober@oceania ~]$ docker run -d --name mynginx nginx
c1c452da55935ef3bcfa14f6f0707cfbc516c1fa6ad1b9e6a4898144945d1922
[rober@oceania ~]$ docker port mynginx
[rober@oceania ~]$ docker stop c1c452da55935ef
c1c452da55935ef
[rober@oceania ~]$ docker rm c1c452da55935ef
c1c452da55935ef
[rober@oceania ~]$ docker run -d --name mynginx -P nginx
abfec277a4810b3dd42ad017347d4cb19c36db0ea23a5ddbecde4295ddf12e53
[rober@oceania ~]$ docker port mynginx
80/tcp -> 0.0.0.0:32770
[rober@oceania ~]$ █
```



# Asignación de puertos estática

73

- También es posible especificar una asignación de puertos específica con la opción **-p** de *docker run*
- En este ejemplo el puerto 80 (TCP) del contenedor se mapea al puerto 80 del host

```
[rober@oceania ~]$ docker rm -f abfec277a4810b3dd42ad  
abfec277a4810b3dd42ad  
[rober@oceania ~]$ docker run -d --name mynginx -p 80:80 nginx  
518737c0d9ed9c4f5dc519303ebce900e586c5adc8189b3f5f36641f6a9d7e53  
[rober@oceania ~]$ docker port mynginx  
80/tcp -> 0.0.0.0:80  
[rober@oceania ~]$ curl localhost  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>
```



# Red de un contenedor

74

- Cada contenedor tiene un *stack* de red propia
  - Configuración independiente de rutas, interfaces, etc
- Conceptualmente una red en Docker es un *switch virtual* que se implementa a través de un **driver de red**
  - Un contenedor puede estar conectado a múltiples redes a través de diferentes drivers
- Un contenedor solo puede “ver” sus interfaces de red
  - Desde el *host* se puede acceder a todos los contenedores
  - Los contenedores no se pueden acceder desde el exterior
- Las redes se comportan como objetos atómicos
  - Se crean y se destruyen
  - Se añaden y se eliminan contenedores dinámicamente



# Redes en Docker

75

- El comando **docker network** permite administrar las redes
  - El argumento `-ls` permite listar las redes disponibles
  - Docker crea tres redes por defecto (ver figura): *bridge*, *host* y *none*
- Se puede seleccionar la red en la que se crea un contenedor con la opción `--net <net-name>` de **docker run**

```
[rober@oceania ~]$ docker network ls
NETWORK ID            NAME      DRIVER      SCOPE
1bde0c32e45d          bridge    bridge      local
9c93c1c1a601          host      host       local
b3554dc448fd          none     null       local
[rober@oceania ~]$
```



# Drivers de red

76

- Docker soporta los siguientes *drivers* de red
  - *bridge* (*driver* por defecto)
  - *host*
  - *overlay*
  - *macvlan*
  - *null*
- *Plugins* de red
  - Son *drivers* de terceros para proporciona conectividad de red
  - Permiten integrar Docker con *stacks* de red especializadas



# Driver bridge

77

- El contenedor tiene dos interfaces de red
  - Una interfaz de *loopback*
  - Una interfaz virtual conectada al *bridge docker0* del host
    - El *bridge docker0* tiene asignada la subred privada 172.17.0.0/16
    - El tráfico se gestiona mediante *iptables*

```
[rober@oceania ~]$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
        inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
              inet6 fe80::42:21ff:fe15:6259 prefixlen 64 scopeid 0x20<link>
                ether 02:42:21:15:62:59 txqueuelen 0 (Ethernet)
                  RX packets 0 bytes 0 (0.0 B)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 7 bytes 801 (801.0 B)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

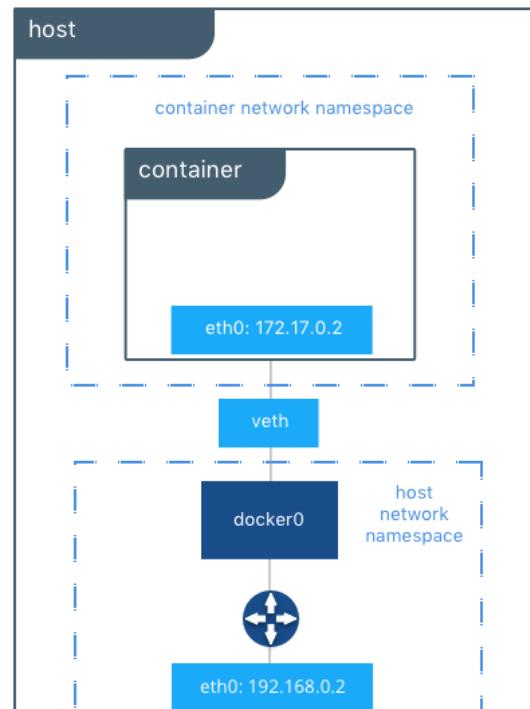
[rober@oceania ~]$ docker run -ti --name utools busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
45: eth0@if46: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
[rober@oceania ~]$ █
```



# Driver bridge

78

- Red *bridge* por defecto con *driver bridge*



- La red *bridge* creada por defecto se llama igual que el *driver*

```
[rober@oceania ~]$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
1bde0c32e45d    bridge    bridge      local
9c93c1c1a601    host      host       local
b3554dc448fd    none     null       local
[rober@oceania ~]$
```



# Driver host

79

- El contenedor comparte la *stack* de red del host
  - Puede ver y administrar todas las interfaces del host
  - El tráfico es directo (no va a través de ningún *bridge*)
    - Rendimiento de red nativo
- Se configura con *docker run --net host*
  - La red host creada por defecto se llama igual que el driver

---

```
[rober@oceania ~]$ docker run -ti --name mynginx --net host busybox ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: en0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:d8:61:15:f9:71 brd ff:ff:ff:ff:ff:ff
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue qlen 1000
    link/ether 52:54:00:3e:72:f5 brd ff:ff:ff:ff:ff:ff
4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 qlen 1000
    link/ether 52:54:00:3e:72:f5 brd ff:ff:ff:ff:ff:ff
5: vboxnet0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:21:15:62:59 brd ff:ff:ff:ff:ff:ff
[rober@oceania ~]$ █
```



# Driver overlay

80

- Permite conectar demonios Docker ejecutándose en diferentes *hosts*
- De esta forma habilita la comunicación segura entre:
  - Contenedores que se ejecutan en diferentes *hosts* (i.e. en diferentes demonios Docker)
  - Servicios creados con Docker Swarm (entorno clúster)
- Por tanto, este *driver* permite crear redes ***multihost***
- Docker maneja de forma transparente el enrutamiento de cada paquete hacia y desde el *host* Docker correcto y el contenedor de destino



# Driver macvlan

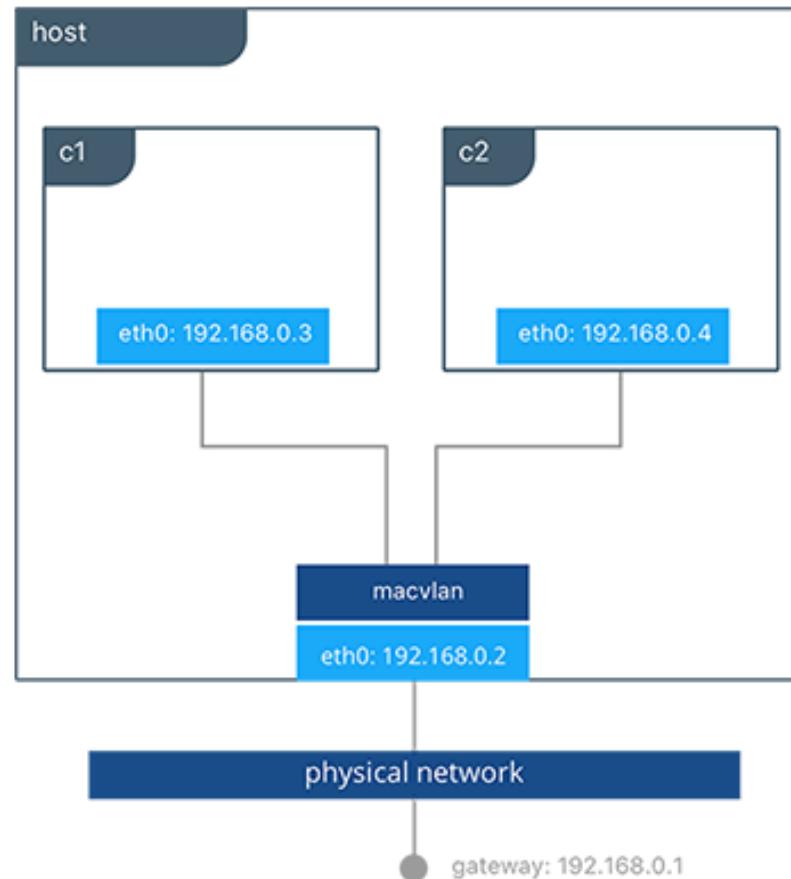
81

- Permite asignar una dirección MAC a cada interfaz de red virtual de un contenedor
  - Esto hace que dicha interfaz parezca una interfaz física directamente conectada a la red
  - El demonio Docker dirige el tráfico a los contenedores por sus direcciones MAC
- Es necesario designar una interfaz de red física del *host* para ser usada con este *driver*, así como la *subred* y la *puerta de enlace*
- Útil cuando se necesita que los contenedores se vean como *hosts* físicos en una red cada uno con una dirección MAC única
  - Ciertas aplicaciones esperan conectarse directamente a la red física en lugar de enrutarse a través del *stack* de red del *host*



# Driver macvlan

82





# Driver null

83

- El contenedor solo posee la interfaz de *loopback*
  - De esta forma, el contenedor está completamente aislado
  - Útil para ejecutar aplicaciones que no requieren ningún tipo de conectividad de red

```
[rober@oceania ~]$ docker run -ti --name mynginx --net none busybox ip link  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
[rober@oceania ~]$ █
```

- Se configura con *docker run --net none*
  - La red creada por defecto con *driver null* se llama *none*

```
[rober@oceania ~]$ docker network ls  
NETWORK ID      NAME      DRIVER      SCOPE  
1bde0c32e45d    bridge    bridge      local  
9c93c1c1a601    host      host       local  
b3554dc448fd    none      null       local  
[rober@oceania ~]$ █
```



# Creación de redes

84

- Docker permite crear nuevas redes y conectar contenedores a ellas
  - Comando **docker network**
- Crear una red llamada “frontend”
  - Por defecto se crea con el *driver bridge* (su scope es local)
  - Existen parámetros de *docker network create* que permiten especificar la subred, puerta de enlace, etc

```
[rober@oceania ~]$ docker network create frontend
d18336ef7afcd17f048a30e07ec4790f9e5b46df1c6e3ef2ac18d155d0c12268
[rober@oceania ~]$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
1bde0c32e45d    bridge    bridge      local
d18336ef7afc    frontend  bridge      local
9c93c1c1a601    host     host       local
b3554dc448fd    none     null       local
[rober@oceania ~]$ █
```



# Creación de redes

85

- Inspeccionamos la red “frontend” con **docker network inspect**

```
[rober@oceania ~]$ docker network inspect frontend
[
    {
        "Name": "frontend",
        "Id": "d18336ef7afcd17f048a30e07ec4790f9e5b46df1c6e3ef2ac18d155d0c12268",
        "Created": "2020-01-15T09:16:16.307826915+01:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```



# Coneectar contenedores a redes

86

- Conectamos un contenedor a la red “frontend”
  - Comando **docker network connect**
- Lo inspeccionamos para comprobar a qué redes está conectado
- Ahora, el contenedor está conectado a dos redes
  - A la red bridge por defecto (conectado a ella en su creación)
  - A la red frontend creada por el usuario

```
[rober@oceania ~]$ docker run -d --name mynginx nginx  
b2bb5eee2275ad125c667f09104a67ac96c34a4c50679cb838c0f3a7d142941f  
[rober@oceania ~]$ docker network connect frontend mynginx  
[rober@oceania ~]$ docker inspect --format="{{.NetworkSettings.Networks}}" mynginx  
map[bridge:0xc000266480 frontend:0xc000266540]  
[rober@oceania ~]$ █
```



# Desconectar y eliminar redes

87

- Desconectamos el contenedor de la red por defecto (*bridge*)
  - Comando **docker network disconnect**
  - Comprobamos qué contenedores están conectados a la red por defecto (ninguno en el ejemplo) y a la red “frontend”
- Eliminamos la red “frontend”
  - Comando **docker network rm**
  - No es posible eliminar una red con contenedores conectados

```
[rober@oceania ~]$ docker network disconnect bridge mynginx
[rober@oceania ~]$ docker network inspect bridge | jq .[].Containers[].Name
[rober@oceania ~]$ docker network inspect frontend | jq .[].Containers[].Name
"mynginx"
[rober@oceania ~]$ █
```

```
[rober@oceania ~]$ docker network rm frontend
Error response from daemon: error while removing network: network frontend id d18336
ef7afcd17f048a30e07ec4790f9e5b46df1c6e3ef2ac18d155d0c12268 has active endpoints
[rober@oceania ~]$ █
```



# Descubrimiento de servicios

88

- Los contenedores pueden comunicarse entre ellos a través de sus direcciones IPs
  - Es posible especificar una IP fija a un contenedor
- Sin embargo, no es cómodo trabajar con IPs directamente
- Los contenedores también pueden referenciarse por su nombre
  - **El nombre de un contenedor Docker es único en el host**
- Docker integra un **servidor DNS** que realiza la resolución de nombres
  - Pero **NO funciona** en las redes creadas por defecto!

---

```
[rober@oceania ~]$ docker run -d --name mynginx nginx
cbd430159142de75de603db9f7d0795e4bf74fb18e0cf74831a768adbf31c5e7
[rober@oceania ~]$ docker inspect --format="{{.NetworkSettings.IPAddress}}" mynginx
172.17.0.2
[rober@oceania ~]$ docker run --name mybusybox busybox ping -c 1 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.100 ms

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.100/0.100/0.100 ms
[rober@oceania ~]$ docker run --name mybusybox2 busybox ping -c 1 mynginx
ping: bad address 'mynginx'
[rober@oceania ~]$ █
```



# Descubrimiento de servicios

89

- Docker integra un servidor DNS que realiza la resolución de nombres
  - Conectamos el contenedor mynginx a nuestra red “frontend”
  - Inspeccionamos la red para obtener la dirección IP del contenedor
  - Comprobamos con ping la conectividad usando su nombre
    - Ahora sí funciona ya que “frontend” es una red definida (creada) por el usuario

```
[rober@oceania ~]$ docker network connect frontend mynginx
[rober@oceania ~]$ docker network inspect frontend | jq .[].IPAM.Config[].Subnet
"172.19.0.0/16"
[rober@oceania ~]$ docker network inspect frontend | jq .[].Containers[].IPv4Address
"172.19.0.2/16"
[rober@oceania ~]$ docker run --name mybusybox3 --net frontend busybox ping -c 1 mynginx
PING mynginx (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.077 ms

--- mynginx ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.077/0.077/0.077 ms
[rober@oceania ~]$ █
```



# Conectividad entre redes

90

- Creamos un contenedor en la red “frontend”
- Creamos otra red con nombre “backend” y un contenedor en dicha red
- Hacemos *ping* desde un contenedor al otro usando la IP
  - **docker exec** permite ejecutar un comando dentro de un contenedor en ejecución
- Contenedores conectados a redes diferentes no pueden comunicarse directamente
  - Esto permite aislar grupos de contenedores
  - Sí podrían comunicarse a través del host mapeando puertos públicos o usando un contenedor intermedio (*proxy*) conectado a ambas redes

```
[rober@oceania ~]$ docker run --rm -dti --name busybox1 --net frontend busybox  
f19c4e239b9c02c5f0bac50b49e21194b5766306e52d452d890bc6543c7959e4  
[rober@oceania ~]$ docker run --rm -dti --name busybox2 --net backend busybox  
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.Networks.frontend.IPAddress}}" busybox1  
172.19.0.2  
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" busybox2  
172.20.0.2  
[rober@oceania ~]$ docker exec busybox1 ping -c 1 172.20.0.2  
PING 172.20.0.2 (172.20.0.2): 56 data bytes  
  
--- 172.20.0.2 ping statistics ---  
1 packets transmitted, 0 packets received, 100% packet loss
```



# Alias de red

91

- El nombre de un contenedor Docker es **único** en el host
  - Creamos un contenedor busybox conectado a la red “frontend”
  - Creamos otro contenedor busybox conectado a la red “backend”
  - Obtenemos un error aunque sus FQDN serían teóricamente distintos (`busybox.frontend != busybox.backend`)

---

```
[rober@oceania ~]$ docker run --rm -dti --name busybox --net frontend busybox
64b60a9d484aa7ddaed029e3bf3baf1cc424a1e7bb8b0cd92fd917973fe250f5
[rober@oceania ~]$ docker run --rm -dti --name busybox --net backend busybox
docker: Error response from daemon: Conflict. The container name "/busybox" is already in use by container "64b60a9d484aa7ddaed029e3bf3baf1cc424a1e7bb8b0cd92fd917973fe250f5". You have to remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
[rober@oceania ~]$ █
```



# Alias de red

92

- Un **alias de red** permite definir un nombre de un contenedor específicamente para una red en concreto
  - Opción `--net-alias` de `docker run`
  - Opción `--alias` de `docker network connect`
  - El nombre solo es válido en la red en la cual se define
  - Los alias en una **misma red no son únicos**

```
[rober@oceania ~]$ docker run --rm -dti --name busybox --net frontend busybox
afb0c689d86f000c739aec9435dd6bb886fbb8eb2b112b1d318b4ba44bfff592
[rober@oceania ~]$ docker run --rm -dti --name busybox2 --net backend --net-alias busybox busybox
b9b669b417a4c6bbcd790a9f921314b5ac19e6211b7e80e2bfcccc502841c6ff
[rober@oceania ~]$ docker run --rm -dti --name busybox3 --net backend --net-alias busybox busybox
5cd9581433bb9a1fc960338ed66606080c2f224ffad5ccbad22f914a6e922a92
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.Networks.frontend.IPAddress}}" busybox
172.19.0.2
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" busybox2
172.20.0.2
[rober@oceania ~]$ docker inspect --format "{{.NetworkSettings.Networks.backend.IPAddress}}" busybox3
172.20.0.3
[rober@oceania ~]$ █
```



# Balanceo de carga

93

- Si hay varios contenedores en la misma red con el mismo alias, el servidor DNS devuelve las direcciones IPs de todos los contenedores que lo usen
- Las direcciones IPs se devuelven en orden *round-robin*
  - Balanceo de carga "simple y barato"

```
[rober@oceania ~]$ docker run --rm -dti --name busybox2 --net backend --net-alias busybox busybox  
c4b933a4eabfb96398197be9a6af3acb5685d0a187ca87eea5fa2a2afb1e3b4c  
^[[A[rober@oceania ~]$ docrun --rm -dti --name busybox3 --net backend --net-alias busybox busybox  
65e91d3a5f2166b7ef122ad85a4e9a2afdbebcaa5e04f41a0cc30598a3b4b189  
[rober@oceania ~]$ docker run --rm --net backend busybox ping -c 1 busybox  
PING busybox (172.20.0.3): 56 data bytes  
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.071 ms  
  
--- busybox ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 0.071/0.071/0.071 ms  
[rober@oceania ~]$ docker run --rm --net backend busybox ping -c 1 busybox  
PING busybox (172.20.0.2): 56 data bytes  
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.084 ms  
  
--- busybox ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 0.084/0.084/0.084 ms
```



# Links

94

- Los *links* añaden al fichero `/etc/hosts` una entrada al contenedor referenciado
  - Opción `--link` de `docker run`
  - Se consideran **legacy** (es recomendable usar redes definidas por el usuario)
- Es posible definir el nombre con el que se referencia a un contenedor
  - En el segundo ejemplo mostrado, `--link mynginx:nginx` significa que se puede referenciar al contenedor llamado `mynginx` con el nombre `nginx`
- No se puede crear un *link* a un contenedor que no está en ejecución

```
[rober@oceania ~]$ docker run -d --name mynginx nginx
13d1da00314dd6b22a3b0ea6bc08bf00d17fa63634d97ebc40c565c15cbe6221
[rober@oceania ~]$ docker run --rm --link mynginx busybox cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      mynginx 13d1da00314d
172.17.0.3      9d150e322838
[rober@oceania ~]$ docker run --rm --link mynginx:nginx busybox cat /etc/hosts
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      nginx 13d1da00314d mynginx
172.17.0.3      9c93786dd534
```



# Contenidos

95

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- **Almacenamiento**
- Docker Compose
- Docker Swarm
- Conceptos básicos de seguridad



# Introducción

96

- Es posible almacenar datos en la capa de escritura de un contenedor, pero dichos datos **no se persisten** cuando el contenedor se detiene
- Escribir en la capa de escritura del contenedor requiere usar un *driver* de almacenamiento (*OverlayFS*) para manejar el sistema de ficheros
  - El *driver* proporciona un sistema de ficheros de tipo "Union"
    - Permiten montar sistemas de ficheros formados por la unión de otros sistemas de ficheros, de forma que se superponen transparentemente formando así un único sistema de ficheros
    - <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>
  - Esta abstracción extra proporcionada por el *driver* reduce el rendimiento
- Docker ofrece tres formas distintas de gestionar el almacenamiento
  - Volúmenes
  - Montajes *bind* (*bind mounts*)
  - Montajes *tmpfs* (*tmpfs mounts*)



# Tipos de almacenamiento

97

## ● Volúmenes

- Se almacenan en la ruta del sistema de ficheros del host que es gestionada por Docker (en Linux sería: `/var/lib/docker/volumes`)
- Otros procesos del sistema **no deberían** modificar dicha ruta
- Se consideran la mejor forma de persistir datos en Docker
- Los volúmenes pueden tener un nombre o ser anónimos

## ● Montajes bind

- Permiten montar un directorio del host en un contenedor ya que se puede usar cualquier ruta del sistema de ficheros del host
- Otros procesos del sistema podrían modificar ficheros en esas rutas en cualquier momento
- Importante: un contenedor podría modificar ficheros del host

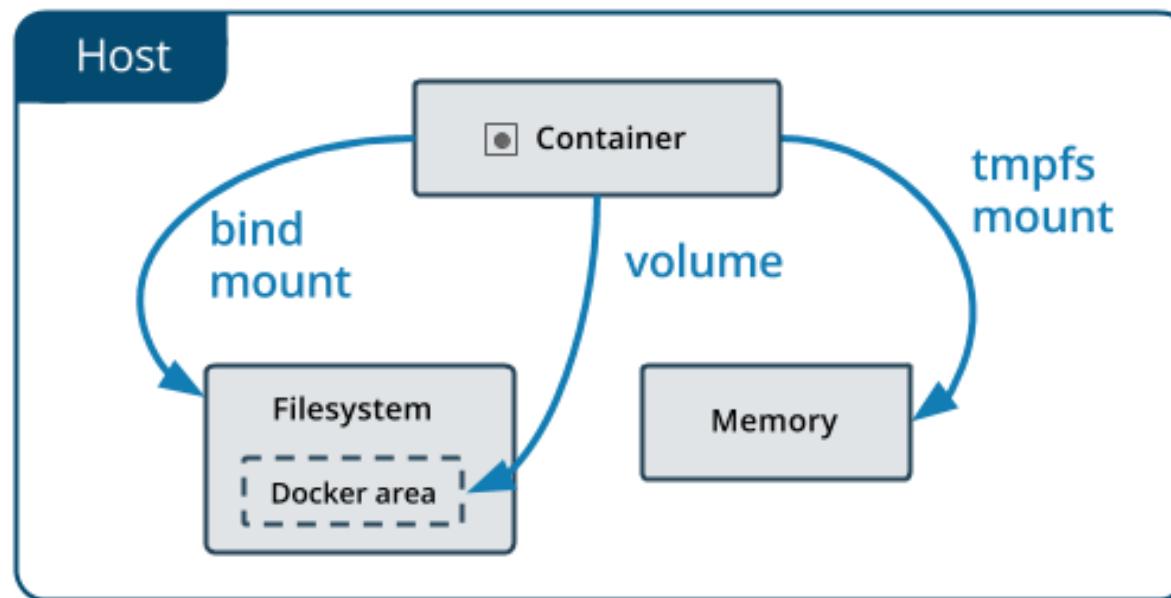
## ● Montajes *tmpfs*

- Se almacenan en la memoria del host y por tanto sus datos nunca son escritos a ningún sistema de ficheros del mismo
- Realmente **no permiten persistir datos en disco**, pero son útiles para datos de estado (no persistentes) evitando así usar la capa de escritura y proporcionando además gran rendimiento



# Tipos de almacenamiento

98





# ¿Qué es un volumen?

99

- Es un directorio que reside en el área del *host* gestionada por Docker y que puede ser montado en un contenedor como un sistema de ficheros adicional o complementario al *rootfs*
- Se puede ver como una forma de compartir directorios y ficheros entre contenedores de forma sencilla
  - **Los datos de un volumen no forman parte de una imagen**
- Permite persistir datos ya que los volúmenes existen de forma **independiente** al ciclo de vida de los contenedores
- Proporcionan mayor rendimiento que usar la capa de escritura
  - Además, no incrementa el tamaño del contenedor que lo utiliza
- **Un volumen puede crearse con un nombre (*named volume*) o ser anónimo**
- Una ventaja sobre los *bind mounts*, es que un volumen es gestionado completamente por Docker y está aislado del resto del *host*
  - Los *bind mounts* dependen de la estructura de directorios del *host*



# Declaración de volúmenes

100

- Con la opción **-v | --volume** del comando *docker run*
  - **-v [<name>:]<container\_dir>**
  - Ejemplo: *docker run -v myvol:/data nginx*
- Con la opción **--mount** del comando *docker run* (preferible)
  - **--mount type=volume, [source=<name>], target=<container\_dir>**
    - Ojo, el parámetro *type* también soporta *bind* y *tmpfs*
  - Ejemplo: *docker run --mount type=volume,source=myvol,target=/data nginx*
- En el fichero *Dockerfile* usando la instrucción **VOLUME**
  - Cuando se lanza un contenedor, *docker run* creará un nuevo volumen con los datos que existan en la imagen en la ruta especificada

```
[rober@oceania ~]$ cat ubuntu/Dockerfile
FROM ubuntu:bionic
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
ENTRYPOINT cat /myvol/greeting
[rober@oceania ~]$ docker run --rm --name mycontainer myubuntu
hello world
[rober@oceania ~]$ █
```



# Administración de volúmenes

101

- Las opciones `-v` | `--mount` de `docker run` crean el volumen si éste no existe cuando se lanza el contenedor
  - Si el contenedor contiene datos en el punto de montaje especificado para el nuevo volumen, éstos se copian en el volumen
- Es posible administrar los volúmenes de forma independiente a los contenedores con el comando **`docker volume`**
  - Listar volúmenes
    - `docker volume ls`
  - Inspeccionar un volumen
    - `docker volume inspect myvol`
  - Inspeccionar volúmenes en una imagen o contenedor
    - `docker inspect --format "{{.Config.Volumes}}"` `myvol`
  - Eliminar volumen
    - `docker volume rm myvol`



# Administración de volúmenes

102

- Es posible administrar los volúmenes de forma independiente a los contenedores con **docker volume**
  - Ejemplo usando un **volumen anónimo**
    - Su nombre (identificador) es autogenerado

```
[rober@oceania ~]$ docker volume ls
DRIVER          VOLUME NAME
[rober@oceania ~]$ docker run -v /data --rm -dti --name mybusybox busybox
050ffe134b1b03f3a221e60c0efa2dbc2892574c0bf57fab04f9a23f6df73bb2
[rober@oceania ~]$ docker volume ls
DRIVER          VOLUME NAME
local           f9782b7e252a5efe9349eb3dbfaeb7da0f7e9aeb7059f217ff7b9d03ebb60554
[rober@oceania ~]$ docker inspect --format "{{.Config.Volumes}}" mybusybox
map[/data:{}]
[rober@oceania ~]$ docker exec mybusybox sh -c "date > /data/today"
[rober@oceania ~]$ docker exec mybusybox sh -c "cat /data/today"
Thu Jan 16 14:08:13 UTC 2020
[rober@oceania ~]$ █
```



# Administración de volúmenes

103

- Es posible administrar los volúmenes de forma independiente a los contenedores con **docker volume**
  - Ejemplo usando un **volumen con nombre**

```
[rober@oceania ~]$ docker volume ls
DRIVER          VOLUME NAME
[rober@oceania ~]$ docker volume create data
data
[rober@oceania ~]$ docker run -v data:/data --rm -dti --name mybusybox busybox
41be416fcf1ecf1096df15f60e25956682b8f0db8787a6158dc4cf6605874907
[rober@oceania ~]$ docker volume ls
DRIVER          VOLUME NAME
local           data
[rober@oceania ~]$ docker exec mybusybox sh -c "date > /data/today"
[rober@oceania ~]$ docker rm -f mybusybox
mybusybox
[rober@oceania ~]$ docker run -v data:/mydata --rm -dti --name mybusybox busybox
fb551c760dc3a3a9748648d75e1aba5f12e80453e93f7bcb87a8dc96658dc3
[rober@oceania ~]$ docker exec mybusybox sh -c "cat /mydata/today"
Thu Jan 16 14:49:10 UTC 2020
[rober@oceania ~]$ █
```



# Montajes bind

104

- Permiten montar un directorio (o fichero) del *host* en un contenedor
- Con la opción **-v | --volume** del comando *docker run*
  - `-v <host_dir>:<container_dir>`
  - **El directorio debe ser una ruta absoluta, de lo contrario se consideraría un volumen con nombre**
  - Ejemplo: `docker run -v /www:/data/www nginx`
- Con la opción **--mount** del comando *docker run* (preferible)
  - `--mount type=bind,source=<host_dir>,target=<container_dir>`
  - Ejemplo: `docker run --mount type=bind,source=/www,target=/data/www nginx`
- En el fichero *Dockerfile* no está permitido definir un montaje *bind*
  - Por motivos obvios de seguridad
- Si el punto de montaje especificado en el contenedor existe, su contenido se oculta y el contenedor solo “ve” los datos del *host*



# Montajes bind

105

- Permiten montar un directorio (o fichero) del *host* en un contenedor
  - Importante: un contenedor puede modificar ficheros del *host*

---

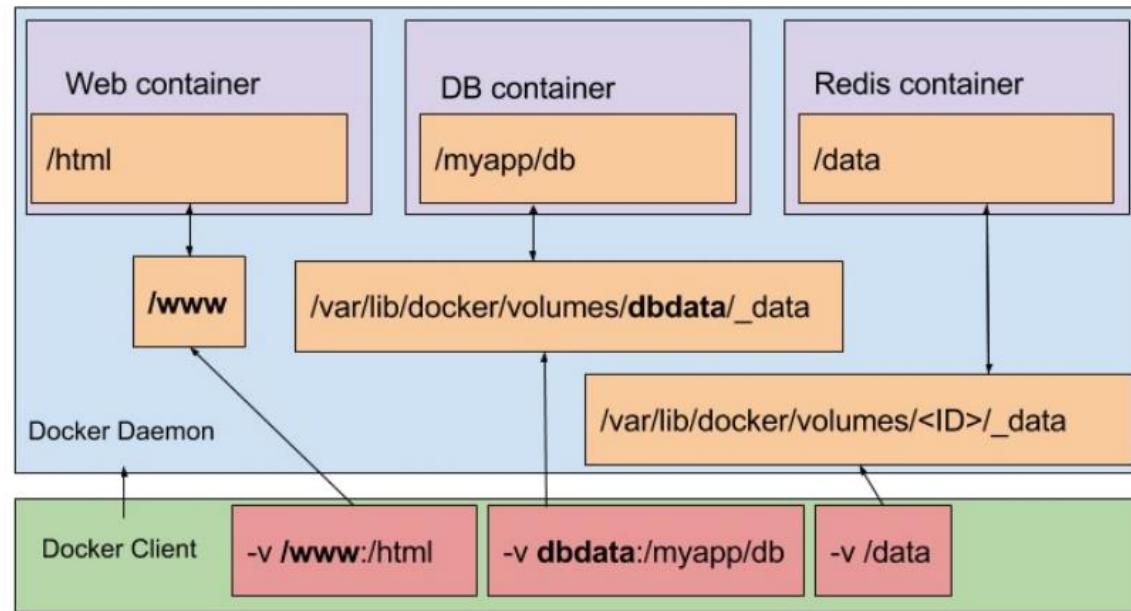
```
[rober@oceania ~]$ ls ubuntu/
Dockerfile
[rober@oceania ~]$ docker run -v /home/rober/ubuntu:/ubuntu --rm -ti --name mybusybox busybox
/ # ls /ubuntu/
Dockerfile
/ # date > /ubuntu/today
/ # exit
[rober@oceania ~]$ cat ubuntu/today
Thu Jan 16 15:12:09 UTC 2020
[rober@oceania ~]$ ls -l ubuntu/
total 8
-rw-rw-r-- 1 rober rober 122 ene 16 13:45 Dockerfile
-rw-r--r-- 1 root root    29 ene 16 16:12 today
[rober@oceania ~]$ █
```



# Comparativa

106

- Montaje *bind* vs volumen con nombre vs volumen anónimo





# Contenidos

107

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- **Docker Compose**
- Docker Swarm
- Conceptos básicos de seguridad



# ¿Qué es Docker Compose?

108

- Es una herramienta que permite definir y ejecutar aplicaciones **multicontenedor** que se ejecutan en el mismo host
  - Una aplicación multicontenedor se denomina **stack** en el ámbito Docker
- Los contenedores (**servicios/microservicios**) que componen la aplicación y sus dependencias se definen en un **fichero en formato YAML**
  - Especifican toda la configuración de cada contenedor
  - Descarga automática de imágenes y ejecución de contenedores
  - Creación de los volúmenes, redes, *links*, etc definidos para todos los contenedores
  - Permiten automatizar el despliegue de aplicaciones multicontenedor (paradigma IaC)
- Usando dicho fichero de configuración, es posible desplegar la aplicación completa mediante un solo comando
  - **docker compose up**



# Sintaxis docker-compose.yml

109

- Existen diferentes versiones de formato para los ficheros compose
  - <https://docs.docker.com/compose/compose-file/compose-versioning>
  - Especificación más reciente
    - <https://docs.docker.com/compose/compose-file/>
  - La versión se puede especificar al inicio del fichero YAML
- En el fichero se definen tres objetos principales
  - **Servicios:** un servicio contiene la configuración que se aplica a cada contenedor que se ejecuta
  - **Redes:** se especifican las redes definidas por el usuario ( adicionales a la red por defecto)
  - **Volúmenes:** para definir volúmenes con nombre
- Similar a los *Dockerfiles*, el fichero *docker-compose.yml* debe residir en una carpeta la cual es usada por Docker Compose como contexto



# Sintaxis docker-compose.yml

110

- La definición de un servicio requiere al menos de una instrucción
  - **build**: indica la carpeta con el *Dockerfile*
  - **image**: define la imagen base
    - Si ambas están presentes, *image* define el tag de la imagen
- Traducción de algunos parámetros de *docker run*
  - **links** se traduce en --link
  - **ports** se traduce en -p
  - **volumes** se traduce en -v
    - En caso de volúmenes con nombre, deben definirse en su propia sección **volumes**
  - **expose** se traduce en --expose
- Además:
  - **command** substituye el CMD del *Dockerfile*
  - **entrypoint** substituye el ENTRYPOINT del *Dockerfile*
  - **healthcheck** substituye el HEALTHCHECK del *Dockerfile*
  - **depends\_on** permite establecer dependencias entre los servicios para controlar el orden en el que se inician y se detienen



# Sintaxis docker-compose.yml

111

- Ejemplo sencillo definiendo un único servicio

```
[rober@oceania compose]$ cat docker-compose.yml
version: "3.7" ← Especificar la versión se considera obsoleto,
services:           se soporta para retrocompatibilidad con
  myapp:             especificaciones previas (2.x y 3.x)
    image: busybox
    command:
      - ls
      - /opt/host_dir
  volumes:
    - type: volume
      source: data
      target: /mydata
    - type: bind
      source: /home/rober/compose
      target: /opt/host_dir
  networks:
    - frontend

  volumes:
    data:

  networks:
    frontend:
```



# Desplegar la aplicación

112

- Comando **docker compose up**
  - Con **docker compose stop** detenemos la aplicación (todos sus contenedores)
  - Con **docker compose down** además de detener la aplicación, elimina los contenedores, redes y volúmenes creados

```
[rober@oceania compose]$ docker-compose up
Creating network "compose_frontend" with the default driver
Creating volume "compose_data" with default driver
Creating compose_myapp_1 ... done
Attaching to compose_myapp_1
myapp_1 | docker-compose.yml
compose_myapp_1 exited with code 0
[rober@oceania compose]$ docker volume ls
DRIVER          VOLUME NAME
local           compose_data
[rober@oceania compose]$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
2d07f55d6e88    backend   bridge      local
1bde0c32e45d    bridge    bridge      local
263172e4f8e7    compose_frontend   bridge      local
d18336ef7afc    frontend   bridge      local
9c93c1c1a601    host      host       local
b3554dc448fd    none      null       local
[rober@oceania compose]$ █
```



En esta figura y siguientes se utilizan comandos **docker-compose** (con guión), pero desde versiones 2.X el comando ha sido renombrado a **docker compose**



# Desplegar la aplicación

113

- Comando **docker compose up**
  - El parámetro `-d` permite el despliegue en segundo plano
  - Podemos ver los logs con el comando **docker compose logs**

---

```
[rober@oceania compose]$ docker-compose up -d
Creating network "compose_frontend" with the default driver
Creating volume "compose_data" with default driver
Creating compose_myapp_1 ... done
[rober@oceania compose]$ docker-compose ps
      Name           Command       State    Ports
----- 
compose_myapp_1   ls /opt/host_dir   Exit 0
[rober@oceania compose]$ docker-compose logs myapp
Attaching to compose_myapp_1
myapp_1 | docker-compose.yml
[rober@oceania compose]$ █
```



# Redes en Docker Compose

114

- Si no se especifica ninguna red, se crea una red por defecto
- Si se especifica una red, los contenedores se conectan únicamente a ella
- Existe conectividad y descubrimiento de servicios entre los contenedores usando como *hostname* el nombre del servicio
  - No se necesitan *links*, pero se pueden usar para definir aliases extra
  - Ejemplo:
    - La aplicación web puede conectarse a postgres://db:5432
    - En el caso derecho, también puede usar postgres://database:5432

```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

```
version: "3"
services:
  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```



# Contenidos

115

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- **Docker Swarm**
- Conceptos básicos de seguridad



# ¿Qué es Docker Swarm?

116

- Es una herramienta que permite definir y ejecutar aplicaciones **multicontenedor** y **multihost** en un clúster de contenedores Docker
  - Por tanto, es una herramienta para la **orquestación de contenedores** Docker en un entorno **clúster**
  - Permite **escalar** las aplicaciones basadas en contenedores en tantas instancias y en tantos nodos de red como se requiera
  - Proporciona **balanceo de carga** y **descubrimiento de servicios**
- Hasta la versión 1.11, Docker Swarm era una herramienta *standalone* independiente de Docker Engine (igual que Docker Compose)
- Desde la versión 1.12, su funcionalidad está integrada de forma nativa en Docker Engine, y se ha denominado **Swarm Mode**
  - La herramienta *standalone* sigue disponible bajo el nombre **Swarm Classic** pero se recomienda el uso de Swarm Mode por su mayor integración y sencillez de uso



# Stack

117

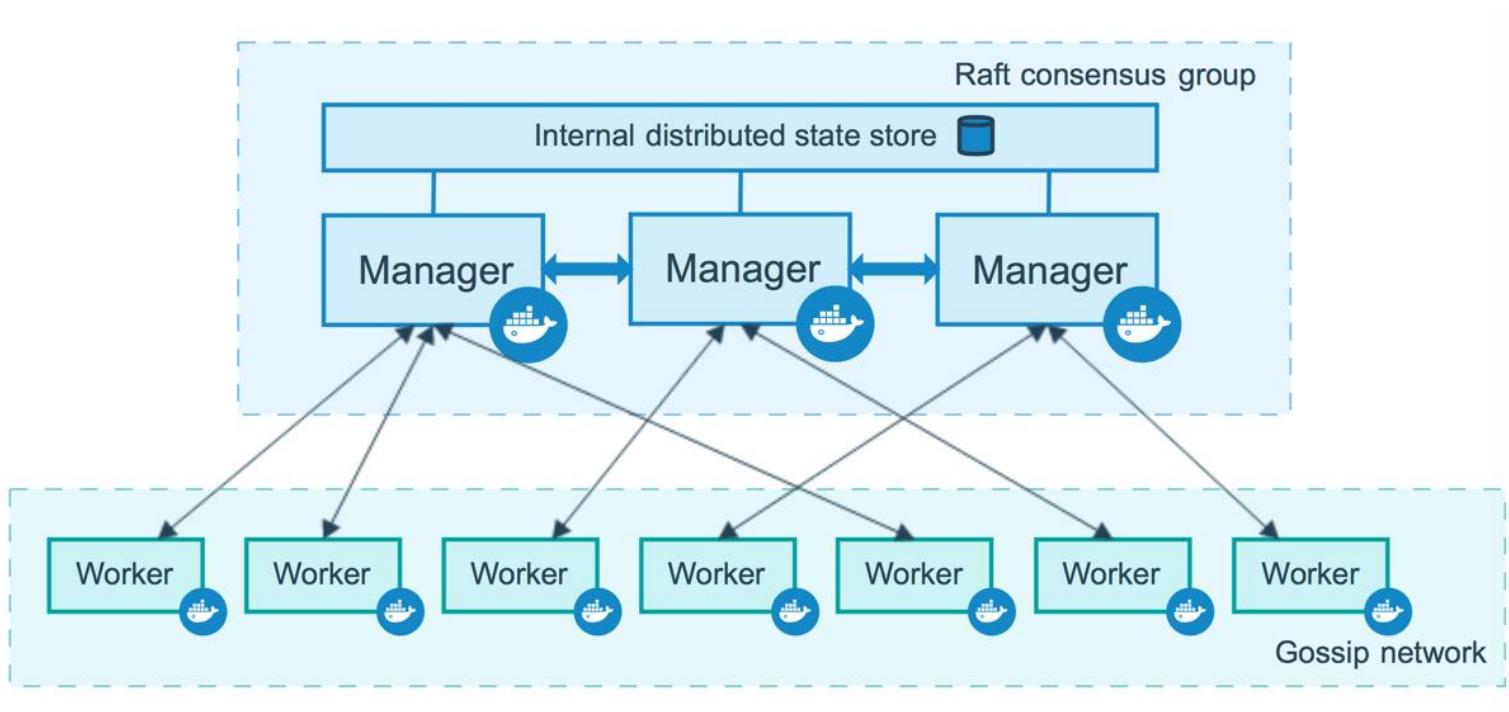
- Un **stack** define un conjunto de servicios posiblemente heterogéneos que componen una aplicación en Swarm
  - **Stack** = aplicación multiservicio
  - Mismo concepto que en Docker Compose pero la aplicación puede ejecutar sus contenedores en múltiples hosts
- Los **servicios** de un **stack** se definen en un **fichero en formato YAML**
  - Prácticamente con la misma sintaxis que los ficheros de Docker Compose pero con pequeñas diferencias
  - El mismo fichero YAML funciona para Docker Compose y Swarm!!
    - Algunas instrucciones son ignoradas por Swarm y otras por Compose
    - Ejemplo: Docker Compose ignora la instrucción **deploy**
- Usando un fichero que define un **stack**, es posible desplegar la aplicación en un clúster Swarm mediante un solo comando
  - **docker deploy stack**



# Arquitectura

118

- Swarm se basa en una arquitectura *master-worker*/*manager-worker*
  - Al menos un nodo maestro, también llamado **manager**
  - Uno o varios nodos **workers**





# Nodos, servicios y tareas

119

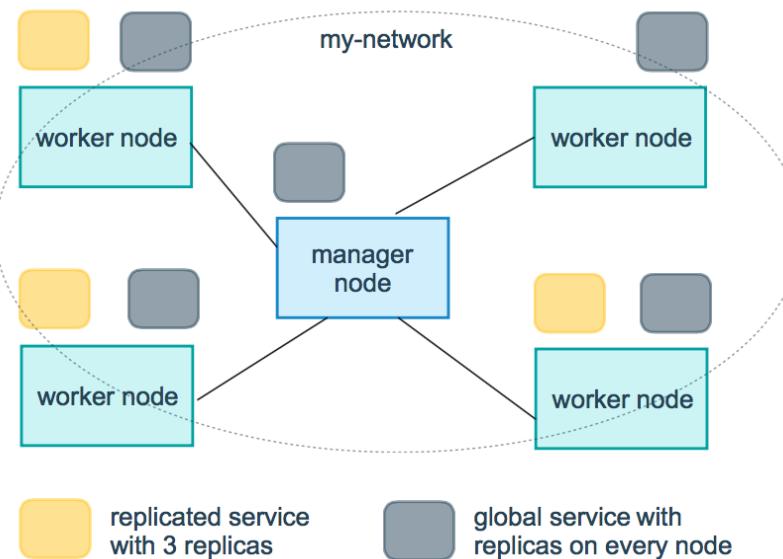
- Un **nodo** es una instancia de Docker Engine que forma parte del clúster Swarm (“el enjambre”)
  - El *manager* recibe la definición de un servicio y la despliega en el clúster
- Un **servicio** en Swarm es una estructura abstracta con la que se pueden definir las tareas individuales a ejecutar en el enjambre
  - Cuando se crea un servicio, el usuario determina la imagen en la que se basa y los comandos que se ejecutan en los contenedores
- Una **tarea** es un contenedor junto con los comandos a ejecutar en él
  - Una tarea es la unidad de trabajo básica en Swarm
- Por tanto, el *manager* es responsable de la gestión del clúster y de la coordinación y planificación de tareas
  - Crea y envía tareas para ser ejecutadas en los nodos *worker*
  - Por defecto, un nodo *manager* actúa como *worker* y también ejecuta tareas, pero esto es configurable



# Servicios replicados y globales

120

- **Replicados:** se trata de tareas que se ejecutan en un número de réplicas (i.e. contenedores) definido por el usuario
  - Los servicios replicados se pueden **escalar** creando réplicas adicionales
- **Globales:** cada nodo del enjambre ejecuta una tarea del servicio correspondiente
  - Si se añade un nuevo nodo, Swarm le asigna una tarea de forma inmediata

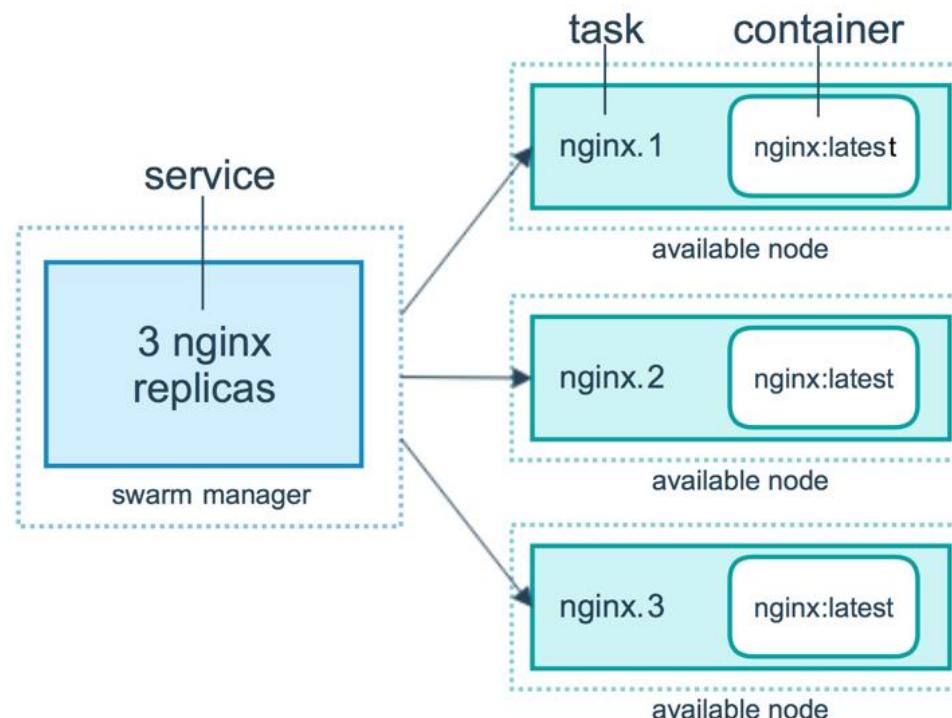




# Servicios replicados y globales

121

- Ejemplo: servicio Nginx replicado con tres instancias
  - Docker distribuye las peticiones entrantes al servicio de forma inteligente entre las instancias disponibles
    - **Balanceo de carga**
  - Cada instancia es una tarea (i.e. contenedor en ejecución)



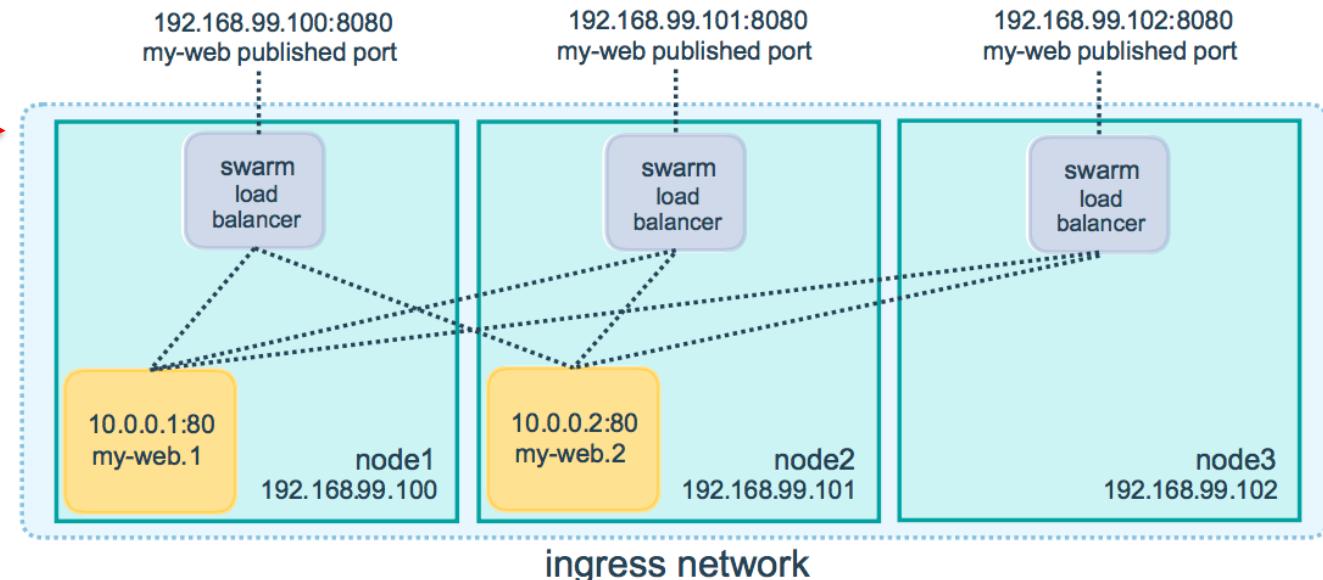


# Ingress routing mesh

122

- Para hacer accesible desde el exterior un servicio desplegado en el clúster, Docker Swarm crea una red con *driver overlay* denominada *ingress*
  - Todos los nodos del clúster pertenecen a la red *ingress*
  - Permite publicar un puerto al exterior y que todos los nodos del clúster acepten peticiones a dicho puerto para cualquier servicio Swarm en ejecución
    - Incluso si no hay un contenedor en ejecución en dicho nodo!
    - Swarm realiza el enrutamiento necesario para encaminar las peticiones al puerto publicado hacia un nodo que disponga de contenedores activos

Cuando se accede al puerto publicado (8080) de cualquier nodo del clúster Swarm, la petición se enruta a un nodo y puerto destino (80) que disponga de un contenedor en ejecución (activo)





# Crear un clúster Swarm

123

- Preparar los nodos del clúster instalando Docker Engine
- Inicializar el clúster Swarm
  - Los nodos con Docker Engine deben ser agrupados en un "enjambre"
  - Primero se debe configurar el nodo que actuará como *manager* ejecutando el comando **docker swarm init** en dicho nodo

Red *ingress* con *driver overlay* (su scope es *swarm*) creada por Docker al inicializar el enjambre

[ rober@oceania ~]\$ docker network ls			
NETWORK ID	NAME	DRIVER	SCOPE
d99843eb61f9	bridge	bridge	local
ad00c81c384e	docker_gwbridge	bridge	local
9c93c1c1a601	host	host	local
frpgnaxi1cpt	ingress	overlay	swarm
b3554dc448fd	none	null	local

- Integrar nodos en el enjambre
  - Al ejecutar el comando anterior, obtendremos por terminal los comandos necesarios para añadir nuevos nodos al clúster Swarm
  - Si no se dispone de dicha información, primero se debe generar el *token* ejecutando en un nodo *manager* el siguiente comando
    - **docker swarm join-token manager | worker**
  - En el nodo que queremos añadir al clúster Swarm, se ejecuta el siguiente comando usando el *token* generado anteriormente
    - **docker swarm join --token [TOKEN]**



# Crear un clúster Swarm

124

- Comprobando el estado del clúster: **docker node**
  - Listar nodos
    - **docker node ls**
  - Inspeccionar un nodo
    - **docker node inspect <node>**
  - Listar tareas en ejecución en un nodo
    - **docker node ps <node>**
  - Eliminar un nodo
    - **docker node rm <node>**

```
[rober@oceania ~]$ docker swarm init
Swarm initialized: current node (lbzv0cyx66lrjpz7eob4mufje) is now a manager.
```

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

```
docker swarm join --token SWMTKN-1-4o8zzvezp814wn1tuxyukelw84c3ds5y9p9e2bde2dzym5ixs3-axlsaj4rb9o5ynflnuu8njef6 193.144.50.55:2377
```

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

```
[rober@oceania ~]$ docker node ls
ID                  HOSTNAME        STATUS        AVAILABILITY      MANAGER STATUS      ENGINE VERSION
lbzv0cyx66lrjpz7eob4mufje *   oceania.des.udc.es  Ready       Active           Leader          19.03.5
[rober@oceania ~]$ 
```



# Definición y despliegue del stack

125

- Definir los servicios de la aplicación mediante un fichero YAML
  - La instrucción **deploy** permite indicar el tipo de servicio (*global* o *replicated*), el número de réplicas, la política de reinicio, límites en el uso de recursos, etc
- Ejecutar un *stack* en el clúster Swarm
  - `docker stack deploy --compose-file stack.yml <stack-name>`
- Listar los *stacks*
  - `docker stack ls`
- Comprobar el estado de los servicios de un *stack*
  - `docker stack services <stack-name>`
- Ver los logs de un servicio
  - `docker service logs <service-name>`
- Escalar servicios replicados
  - `docker service scale <service-name>=<replicas>`
- Detener un *stack*
  - `docker stack rm <stack-name>`



# Ejemplo de fichero stack

126

- Aplicación con dos servicios replicados: wordpress y db

```
[rober@oceania ~]$ cat swarm/stack.yml
version: '3.7'

services:
  wordpress:
    image: wordpress
    ports:
      - "8000:80"
    depends_on:
      - db
    deploy:
      mode: replicated
      replicas: 2

  db:
    image: mysql
    deploy:
      mode: replicated
      replicas: 2
    volumes:
      - db-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress

volumes:
  db-data:
```



# Despliegue del stack

127

```
[rober@oceania ~]$ docker stack deploy --compose-file swarm/stack.yml mystack
Creating network mystack_default
Creating service mystack_db
Creating service mystack_wordpress
[rober@oceania ~]$ docker stack ls
NAME          SERVICES      ORCHESTRATOR
mystack       2            Swarm
[rober@oceania ~]$ docker stack services mystack
ID           NAME          MODE        REPLICAS      IMAGE          PORTS
9aqog10txo9z  mystack_db   replicated  2/2          mysql:latest
sdmyidh7agno  mystack_wordpress  replicated  2/2          wordpress:latest *:8000->80/tcp
[rober@oceania ~]$ 
```



# Despliegue de servicios

128

- Para el despliegue de servicios Docker de **forma individual** en un clúster Swarm se puede usar el comando **docker service**
- Este comando acepta como parámetros muchas de las instrucciones que contienen los ficheros *stack*
- Ejemplo: servicio web nginx replicado con 3 instancias
  - `docker service create --name mynginx --replicas 3 -p 80:80 nginx`

```
[rober@oceania aisi]$ docker service create --name mynginx --replicas 3 -p 80:80 nginx  
tzolgh3joa18x7a9axe08eazs
```

```
overall progress: 3 out of 3 tasks
```

```
1/3: running [=====>]  
2/3: running [=====>]  
3/3: running [=====>]
```

```
verify: Service converged
```

```
[rober@oceania aisi]$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
tzolgh3joa18	mynginx	replicated	3/3	nginx:latest	*:80->80/tcp

```
[rober@oceania aisi]$ docker service ps mynginx
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
ff0je1829si1	mynginx.1	nginx:latest	oceania.des.udc.es	Running	Running about a minute ago		
56xbm3fb57h4	mynginx.2	nginx:latest	oceania.des.udc.es	Running	Running about a minute ago		
hlt7p59rs0t7	mynginx.3	nginx:latest	oceania.des.udc.es	Running	Running about a minute ago		



# Escalado de servicios

129

- Es posible escalar **servicios replicados** de forma dinámica mientras están en ejecución con el comando **docker service scale**
- Este comando acepta como parámetros el nombre del servicio a escalar y el número de réplicas deseado
  - Es posible escalar hacia arriba y hacia abajo el número de instancias de un servicio
- Ejemplo:
  - Escalar el servicio web nginx replicado del ejemplo anterior para que pase a ejecutarse con 5 instancias
    - `docker service scale mynginx=5`



# Contenidos

130

- Introducción
- Conceptos básicos
- Imágenes
- Dockerfiles
- Redes
- Almacenamiento
- Docker Compose
- Docker Swarm
- **Conceptos básicos de seguridad**



# Acceso al API de Docker

131

- Tener acceso al demonio de Docker es como ser *root* en el host
  - **El usuario por defecto en un contenedor es root**
  - Conviene **ejecutar los contenedores** con usuarios no privilegiados
    - Instrucción USER del *Dockerfile* cambia el usuario por defecto
    - Se puede cambiar el UID del *root* dentro de un contenedor (y de otros usuarios) con la opción del demonio Docker: --userns-remap
      - <https://docs.docker.com/engine/security/userns-remap>
- Desde la versión 19.03, Docker soporta de forma experimental un modo *rootless* que **permite a usuarios no privilegiados ejecutar tanto el demonio de Docker como los contenedores (ambos)**
  - Con la opción --userns-remap, el demonio se sigue ejecutando con privilegios de *root*
  - Desde la versión 20.10, el modo *rootless* considera estable
  - Más información sobre el modo *rootless*:
    - <https://docs.docker.com/engine/security/rootless>



# Acceso al API de Docker

132

## ● Ejemplo

- Montamos la raíz del host en un contenedor mediante un montaje *bind*
- Accedemos a información sensible del host
- Creamos un fichero en una ruta sensible
- Podríamos instalar software e incluso apagar la máquina

---

```
[rober@oceania ~]$ head /etc/sudoers
head: no se puede abrir «/etc/sudoers» para lectura: Permisos denegados
[rober@oceania ~]$ docker run --rm -v /:/hostfs --name mybusybox busybox head /hostfs/etc/sudoers
## Sudoers allows particular users to run various commands as
## the root user, without needing the root password.
##
## Examples are provided at the bottom of the file for collections
## of related commands, which can then be delegated out to particular
## users or groups.
##
## This file must be edited with the 'visudo' command.

## Host Aliases
[rober@oceania ~]$ touch /hello.txt
touch: no se puede efectuar `touch` sobre «/hello.txt»: Permisos denegados
[rober@oceania ~]$ docker run --rm -v /:/hostfs --name mybusybox busybox touch /hostfs/hello.txt
[rober@oceania ~]$ ls -l /hello.txt
-rw-r--r-- 1 root root 0 ene 17 13:45 /hello.txt
[rober@oceania ~]$ █
```