Docker para DevOps:
de noob a experto

Iván Martínez - Alvaro Iradier

Docker para DevOps

de noob a experto

Primera edición: septiembre de 2021

Portada: Gerardo Katssenian.

Autores: Álvaro Iradier, Iván Martínez

ÍNDICE

Índice de tablas

Índice de figuras

Bienvenida

Bienvenido/a a esta guía sobre Docker,

Docker es una herramienta muy popular en los equipos desarrollo de software ya que permite simplificar los procesos de programación, despliegue y entrega de aplicaciones. Docker puede resultar de gran utilidad a programadores y administradores de software que centren su actividad en DevOps ya que es una herramienta que mejora la eficiencia de su trabajo al presentarse como una alternativa a la virtualización tradicional.

Esta guía resulta útil tanto a principiantes, como a profesionales que desean seguir aprendiendo. Es recomendable disponer de conocimiento básico en administración y sistemas y programación.

Con esta guía aprenderás qué es Docker, las nociones básicas para empezar a trabajar con esta tecnología, y profundizarás para convertirte en un experto en la creación y configuración de contenedores, así como el despliegue y uso de forma segura, que evite cualquier punto de ataque por un usuario malicioso.

Instalación de Docker

Antes de meternos de lleno en el mundo de Docker y comenzar a comprender y controlar todas las funcionalidades que este nos puede ofrecer, se propone seguir los siguientes pasos:

1. Instalar en el equipo las herramientas VirtualBox, Git y Vagrant.

Virtual Box (versión 6.0)	Git	Vagrant
https://www.virtualbox.org/	https://git-scm.com/	https://www.vagrantup.com/downloads.html

2. Proceder a la instalación de nuestro entorno de pruebas. Para ello, se recomienda acceder al sitio web de GitHub y seguir los pasos descritos en el mismo. Podremos acceder desde la siguiente URL:

   https://github.com/shokone/Vagrant-Docker

3. Y descargar el repositorio desde GitHub, escribiendo para ello el siguiente comando directamente en el editor de git que se abrirá automáticamente tras su instalación:

```
git clone https://github.com/shokone/Vagrant-Docker.git
```

4. Una vez tengamos levantando nuestro entorno de trabajo dejaremos, desde ya, preparada la Máquina Virtual Vagrant, instalando algunas herramientas que usaremos y estudiaremos posteriormente (como cgroups BTRFS). Para ello, ejecutaremos el siguiente comando:

```
$ sudo apt-get install jq cgroup-tools bridge-utils btrfs-tools
```

> El comando `sudo` (super-user-do) es una utilidad que permite a los usuarios ejecutar programas con los privilegios del usuario root (superusuario). Esto es más seguro que iniciar sesión como root, ya que permite conceder estos privilegios a usuarios sin que éstos tengan que conocer la contraseña de root. [1]

5. También debemos instalar Docker Machine, siguiendo las instrucciones detalladas en este enlace:

https://docs.docker.com/machine/install-machine/

CAPÍTULO 1

Introducción a Docker

Introducción a Docker

Qué es Docker

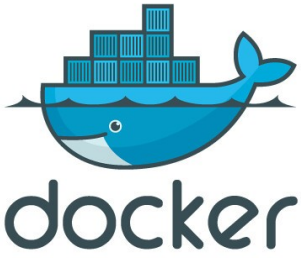

Figura 1: Logo de Docker

Docker es un proyecto de código abierto, cuya finalidad es automatizar el despliegue de aplicaciones informáticas dentro de contenedores de software.

> Contenedores de Software: Aplicaciones agrupadas y aisladas entre sí que se ejecutan sobre un mismo núcleo de sistema operativo. Estas aplicaciones utilizan el sistema operativo de su host en lugar de proporcionar uno propio. [2]

Esto proporciona una capa adicional de abstracción y automatización de la virtualización y ejecución de aplicaciones en múltiples sistemas operativos sin la necesidad de realizar instalaciones complejas.

> Virtualización: En Informática, la virtualización es la creación a través de software de una versión virtual de algún recurso tecnológico, como puede ser una plataforma de hardware, un sistema operativo, un dispositivo de almacenamiento o cualquier otro recurso de red. [3]
>
> En concreto, la virtualización de aplicaciones es la tecnología que permite a un usuario acceder y usar en su ordenador una aplicación instalada en otro ordenador distinto. De la misma manera, un administrador podrá configurar aplicaciones remotas en un servidor y, a continuación, entregar las aplicaciones en el ordenador de un usuario final, siendo la experiencia de uso de la aplicación virtualizada, para el usuario, la misma que si se utilizara la aplicación instalada en una máquina física. [4]

Además, utiliza características propias de aislamiento de recursos del núcleo (o *kernel*) Linux, como pueden ser cgroups [] o namespaces [] permitiendo que contenedores independientes se ejecuten en una única instancia en Linux. Esto evita la sobrecarga del mismo y tener que mantener instaladas diferentes ***máquinas virtuales,*** con el fin de poder correr diferentes versiones de un mismo

software.

> Cgroups: o grupos de control, son una característica del *kernel* de Linux que permiten definir jerarquías en las que se agrupan los procesos de manera que un administrador puede elegir con detalle la manera en la que se asignan los recursos [5]
>
> Namespace: o espacio de nombres es un contenedor abstracto en el que un grupo de uno o más identificadores únicos pueden existir.

Es decir, Docker es una herramienta que puede empaquetar una aplicación y sus dependencias en un contenedor virtual que se puede ejecutar en cualquier servidor Linux. Esto ayuda a permitir la flexibilidad y portabilidad en el medio donde la aplicación se puede ejecutar, ya sea en instalaciones físicas, la nube pública o privada, etc. [6].

Por ejemplo, si dos desarrolladores tienen una aplicación, uno en Java versión 7 y el otro en la versión 8. En caso de que uno de ellos necesite realizar una prueba sobre la aplicación en una versión diferente sobre la cual trabaja, tan solo necesitará levantar un contenedor para poder realizar sus pruebas sin necesidad de instalar una nueva versión de Java en su sistema.

Contenedores vs Máquinas Virtuales

La comparación de un **contenedor** con una **máquina virtual** es algo muy complejo debido, entre otras cosas, a las grandes diferencias esenciales que existen entre ambos conceptos. Estudiaremos primero cada uno de ellos por separado.

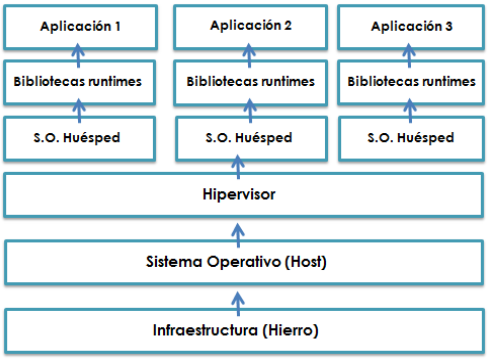

Figura 2: Arquitectura de una máquina virtual

Una **máquina virtual** consiste en la virtualización de un hardware, sobre el que habitualmente se ejecuta un sistema operativo completo que funciona de forma aislada sobre otro sistema operativo completo.

Esta tecnología permite compartir el hardware de tal forma que pueda ser utilizado por varias máquinas virtuales al mismo tiempo. Un esquema simplificado de su arquitectura podría ser el de la Figura 2.

Comenzamos a leerlo de abajo a arriba. Obviamente, lo primero siempre será una máquina física, es decir, algún tipo de hardware que se encargue de sustentarlo todo. A este servidor físico o infraestructura se le conoce coloquialmente en el sector como «***hierro***». Este puede ser un ordenador personal o incluso una máquina ubicada en nuestro centro de procesamiento de datos (CPD).

Pero la infraestructura por si misma no es nada, si no le añadimos un cerebro, es decir, un Sistema Operativo también llamado «host» (ya sea Linux o Windows; también MacOS, aunque es menos común utilizarlo en entornos de producción).

> Host: El término host, o *anfitrión*, se refiere a los ordenadores u otros dispositivos que ofrecen servicios al resto de ordenadores conectados a la red, sea esta local o global como Internet. Es decir, un host es todo equipo informático que posee una dirección IP y que se encuentra interconectado con uno o más equipos y que funciona como el punto de inicio y final de las transferencias de datos. [7]

Posteriormente se encontraría el «hipervisor», que es el software especializado en exponer los recursos hardware del host de modo que puedan ser utilizados por el resto de sistemas operativos. Ejemplos de hipervisores son Virtualbox o Vmware, entre otros.

Y, por último, por encima de todo esto se encontrarían las diferentes aplicaciones y sus **bibliotecas *runtime***, las cuales necesitan disponer de su propio sistema operativo para poder funcionar, y de una parte del hardware utilizado por el host.

> Bibliotecas runtime: Colección de funciones soportadas por un programa cuando este está en ejecución. Estas funciones suministran diferentes servicios de utilidad a dicho programa. [8]

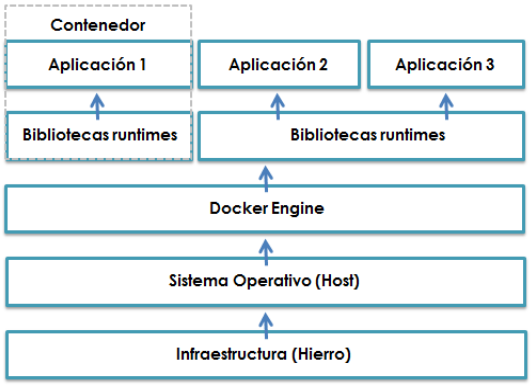

Figura 3: Arquitectura con contenedores en Docker

Por su parte, los **contenedores** también aíslan las aplicaciones y generan un entorno replicable y estable, pero en lugar de albergar un sistema operativo completo, lo hacen compartiendo los recursos del propio host sobre el que se ejecutan. Un esquema simplificado de su arquitectura podría ser el de la Figura 3.

Como se puede observar, los contenedores también necesitan una máquina física, así como un sistema operativo instalado en el host sobre el cual se ejecutan, pero a diferencia de las máquinas virtuales, no necesitan de un hipervisor que corra los diferentes sistemas operativos.

De esto se encargaría lo que en el esquema hemos denominado ***Docker Engine***. Esta herramienta nos permitirá lanzar y gestionar los contenedores con nuestras aplicaciones, pero en lugar de exponer los distintos recursos de la máquina anfitriona, se podrán compartir los mismos entre todos los contenedores. De esta manera se optimiza su uso y se elimina la necesidad de tener sistemas operativos separados para conseguir aislarlos.

Docker utiliza **imágenes** reutilizables entre varias aplicaciones. Cada una de estas se puede asimilar a una capa que puede superponerse a otras para formar un sistema de archivos, que será la combinación de todas ellas. Cuando se lanzan uno o varios contenedores a partir de una misma imagen, estos son aislados de cualquier otra aplicación existente en la máquina, aunque en realidad están compartiendo el mismo sistema operativo que las alberga a todas.

> Imagen: Las imágenes Docker son plantillas de solo lectura en forma de archivo de texto que utiliza el motor de Docker para construir un contenedor y ejecutarlo. Se puede considerar, así, que una imagen de Docker es esencialmente una instantánea de un contenedor. [9] [10]

RECUERDA

Contenedores

- No es necesario instalar un sistema operativo completo por contenedor.
- Su consumo de recursos es muy liviano.
- Se pueden lanzar muchos contenedores al mismo tiempo.
- Por defecto, no disponen de almacenamiento persistente.
- Utilizan una vista aislada de un adaptador de red.

Máquinas virtuales

- Tienen su propio sistema operativo completo.
- Consumen muchos recursos.
- Un equipo no es capaz de mantener encendidas muchas maquinas virtuales.
- Parten de un almacenamiento persistente.
- Utilizan adaptadores de red virtuales.

En definitiva, Docker se trata de un método de virtualización que se integra al nivel del Sistema Operativo para implementar y ejecutar aplicaciones sin tener que instalar una maquina virtual completa para cada aplicación. Para gestionar todos los contenedores que pueden llegar a existir, se debe habilitar un punto central de control, o host con el que todos los contenedores se comunican y desde donde son gestionados.

Historia de la contenedorización

Desde el surgimiento del concepto de *Chroot* hasta la existencia de Docker tal y como lo conocemos a día de hoy, se han producido multitud de aportaciones a esta tecnología por parte de diferentes compañías y entidades. En la siguiente línea histórica hemos querido reflejar algunos de estos eventos y aportaciones para que el lector se pueda hacer a la idea de la complejidad que hay detrás de la creación de esta potente herramienta. ¡Empecemos!

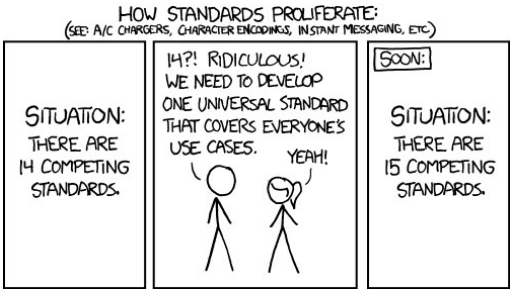

Figura 4: Viñeta sobre la estandarización [Comic n927 de la web xkcd.com]

- 1979 - **Chroot**: Un mecanismo para cambiar el directorio raíz de un proceso y sus hijos, y por tanto ejecutarlo en un “chroot jail”. Proporciona un cierto aislamiento, aunque si el usuario es root, es relativamente trivial escapar de la jaula.
- 1999 - **FreeBSD Jail:** permite trocear un sistema FreeBSD en varios mini-sistemas llamados “Jails”. No solo crea una jaula del sistema de archivos,también crea su propio subsistema de red y dirección IP, impide interaccionar con otros procesos, cargar módulos del kernel, montar sistemas de archivos, etc.
- 2001 - **Linux VServers**, mediante parches de kernel, provee aislamiento de sistema de archivos, tiempo de CPU, direcciones de red y memoria.

- 2002 – Lanzamiento al público del proyecto Open Virtuozzo (**Open VZ**) aunque su historia empezó en 1999 (enlace para saber más)[1]
- 2005 – Lanzamiento al público del proyecto **Solaris Zones y Solaris Containers**
- 2006 – Se desarrolla el concepto de "process containers", renombrada a "**cgroups**- control groups" en 2007, incluído en el kernel Linux en 2008. Permite agrupar procesos compartiendo memoria, CPU y sistema de archivos.
- 2008 – LXC (Linux Containers), desarrollado por IBM, combina cgroups y aislamiento mediante namespaces, además de Chroot. **No** requiere **parches adicionales en el kernel**, y utiliza:
    - Kernel namespaces (ipc, uts, mount, pid, network and user)
    - Apparmor and SELinux profiles
    - Seccomp policies
    - Chroots (using pivot_root)
    - Kernel capabilities
    - CGroups (control groups)
- 2010-2013 – Desarrollo de las utilidades systemd-nspawn y machinectl
- 2013 - **Docker nace**. Utilizaría LXC en sus orígenes.
- 2013 – Nace el lmctfy de Google, de su proyecto Borg. Más tarde se uniría con Docker, finalmente se paraliza su desarrollo en 2015.
- 2014 (Marzo) - Docker migra a libcontainer.
- 2014 (Diciembre) – **Nace** App Container (**appC)** para el sistema operativo CoreOS, especificación abierta, usada por el software rkt. Esta define:
    - Entorno de ejecución

1 https://wiki.openvz.org/History#1999

  - Formato de imagen
  - Protocolo de descubrimiento

- Aunque **appC** no se ha extendido demasiado, y rkt está migrando para soportar OCI (Open Container Initiative), cumplió su papel al desarrollarse como un estándar abierto desde el principio y lograr que se crearan estándares abiertos para la comunidad.
- 2015 - Nace la **CNCF** (Cloud Native Computing Foundation), anunciada junto al proyecto Kubernetes 1.0, con el objetivo de ayudar al avance de las tecnologías de contenedores.
  - Fundadores: Google, CoreOS, Mesosphere, Red Hat, Twitter, Huawei, Intel, Cisco, IBM, Docker, Univa, and VMware.
  - En la actualidad, multitud de miembros y proyectos.
- 2015 (Junio) - Docker anuncia Open Container Initiative (OCI[2])
  - Runtime-spec (runC, implementación de referencia de *containerd*)
  - Image-spec
- 2015 (Julio) - Docker migra a runC (standard OCF - Open Container Format)
- 2016 – Nace *containerd*, al romper la estructura monolítica de Docker
- 2017 - containerd es “aceptado” en la CNCF
- 2018 – La empresa de software Redhat adquiere CoreOS y rkt
- 2019 - containerd “graduado” en la CNCF

Breve Historia de Docker

Docker fue desarrollada como *platform-as-a-service (PaaS)* por la compañía DotCloud, que la usaba para ejecutar las aplicaciones web de sus usuarios en

2 Actualmente la OCI tiene, aparte de Docker, miembros como Google o CoreOS (creadores de rkt)

contenedores, tipo Heroku, Cloud Foundry, Openshift.

DotCloud comenzó a usar AUFS en 2008 (hablaremos de AUFS y Copy-On-Write más adelante) como una forma de optimizar el uso de espacio en disco, junto a tecnologías como Vserver, luego OpenVZ, y finalmente LXC.

> Platform-as-a-service (PaaS): En castellano, *plataforma como servicio*, es un entorno de desarrollo e implementación completo en la nube, a través de cual el proveedor proporciona al cliente un entorno de desarrollo y el paquete de herramientas necesarias para el desarrollo de nuevas aplicaciones. Así, se «alquilan» los recursos que se necesiten a un proveedor de servicios, a los que se accede a través de Internet, y solo se paga por el uso que haga de ellos. [11]

¿Y por qué Docker?

¿Qué proporciona Docker?

- Un **formato** de imagen de contenedores.
- Un método para **construir** las imágenes (con el fichero Dockerfile o el comando `docker build`)
- Una forma de **gestionar** las **imágenes** (con los comandos `docker images`, `docker rmi`, etc.)
- Una forma de **gestionar** los **contenedores** (con los comandos `docker ps`, `docker rm`, etc.)
- Una forma de **compartir** las imágenes (con los comandos `docker push`/`pull`)
- Una forma de **ejecutar** las imágenes (con el comando `docker run`)

¿Cuáles son las claves que hacen tan diferencial esta tecnología?

- Docker permite crear imágenes autocontenidas, con una aplicación (o varias, aunque no debería ser el caso) y todas sus dependencias, bibliotecas, paquetes, etc. Múltiples instancias de estas imágenes se pueden desplegar de forma muy sencilla, en forma de contenedores, ya sea en un entorno local de desarrollo, en un servidor On-Prem (*on-premise,* instalado localmente dentro del servidor y la infraestructura de la propia empresa), o en la nube, sin ningún cambio.
- La virtualización de Docker es muy ligera. Aprovecha los recursos de la máquina de forma muy efectiva, con muy poca sobrecarga, y permite lanzar un contenedor en segundos, o incluso menos, a la vez que mantener un alto nivel de aislamiento entre contenedores y los recursos por los que compiten.
- Docker provee un DSL (Domain Specific Language) muy sencillo y potente, usando los Dockerfile para crear imágenes.

- La distribución de las imágenes se hace de forma muy eficiente, re-utilizando capas compartidas entre distintas imágenes para optimizar la transferencia y el espacio utilizado.

¿Qué **ventajas** ofrece Docker frente a otro tipo de tecnologías similares como gestores de paquetes, herramientas de gestión de configuración y máquinas virtuales?:

- **Gestores de paquetes**, tipo RPM / DPKG, etc. Crear una imagen Docker es, en cierto modo, similar a usar un gestor de paquetes. El problema con los gestores de paquetes es que distribuir una aplicación sin sus dependencias, y delegar estas a los gestores de paquetes, suele causar problemas con las bibliotecas compartidas y versiones no compatibles (se quedan anticuadas, distintas distribuciones usan distintas versiones y sistemas de empaquetado distintos, etc.).
- **Herramientas de gestión de configuració**n, como Puppet, Chef, Ansible, Terraform… Estas herramientas permiten gestionar y automatizar la infraestructura, la configuración de los sistemas operativos, etc., para garantizar que el entorno donde desplegamos la aplicación siempre es igual y replicable. Con Docker esto no es necesario, ya que una imagen Docker es autocontenida, y se ejecutará exactamente igual en distintos entornos y Sistemas Operativos (la única diferencia será el kernel, que no debería afectar a no ser que tengamos funcionalidades muy específicas, dependencia de drivers, GPUs, etc).
- **Máquinas virtuales.** Otra tendencia ha sido crear *bundles* o imágenes para tecnologías VMs (Virtual Machines), como VMWare, Hyper-V, VirtualBox, etc. y estándares como OVF. El concepto, de nuevo, es similar: empaquetar la aplicación con todas sus dependencias. El problema es que una máquina virtual emula solo el hardware, por tanto, todas las dependencias necesitan también incluir todo el sistema operativo. Aquí, la ventaja de Docker, es su ligereza (a cambio perdemos un poco en seguridad y aislamiento, como veremos más adelante).

CAPÍTULO 2

Conociendo Docker

Conociendo Docker

Primer Contenedor: Hello-World

En los anteriores capítulos hemos realizado la instalación de Docker, y comprendido los fundamentos de esta herramienta. En este capítulo, vamos a lanzar un primer contenedor a modo de prueba, solo para cerciorarnos de que la instalación se realizó correctamente. Lo haremos en base a la imagen de «**hello-world**» facilitada por Docker, siguiendo los siguientes pasos.

1. Visualizaremos las imágenes existentes en nuestra máquina, introduciendo el siguiente comando en la línea de comandos de Docker (CLI):

```
docker image ls
```

2. Posteriormente, lanzamos nuestro contenedor con el comando:

```
docker run hello-world
```

3. El comando anterior nos mostrará un mensaje como el que vemos a continuación, confirmándonos así que la instalación se realizó según lo esperado:

```
Hello from Docker!
```

Comandos básicos y utilidades

Después comprobar que la instalación se realizó correctamente tras haber lanzado el contenedor «`hello-world`», y para seguir introduciéndonos en el mundo Docker, estudiaremos ahora los comandos básicos de los que esta herramienta dispone.

Para ello, recomendamos seguir la explicación que nos proporciona la página oficial de Docker, a la cual podemos acceder a través del siguiente enlace:

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

CAPÍTULO 3

Dockerfile

Dockerfile

¿Qué es Dockerfile?

En este epígrafe estudiaremos el concepto de **Dockerfile**, por ser este fundamental para seguir explorando todas las posibilidades de Docker.

Para abordar este concepto es importante comprender que en Docker se utilizan imágenes que posteriormente se convierten en contenedores y que las imágenes serán, finalmente, las aplicaciones que utilicemos ya sea en el trabajo diario en la empresa o en cualquier ámbito para el que se desee utilizar esta tecnología.

Figura 5: Secuencia de construcción de un contenedor

Pues bien, un **Dockerfile** es, sencillamente, el fichero de texto plano que contiene las instrucciones necesarias para crear esta imagen, las cuales se expresarán en forma de comandos (por ejemplo, `FROM`, `COPY`, `RUN`, …) [Figura 5]

Algunos de los **comandos** utilizados en el fichero Dockerfile para su configuración, son los siguientes expresados en la Tabla 1: Tabla de comandos.

Tabla 1: Tabla de comandos

Comando	Descripción
`FROM <imagen>:<tag>`	Indica la imagen base que se va a utilizar para seguir futuras instrucciones. El tag es opcional, en caso de no indicarlo cogerá el valor por defecto **«latest»**
`LABEL maintainer=<mail>`	Esta instrucción permite configurar los datos del autor que genera la imagen
`RUN <comando>`	Esta instrucción ejecuta cualquier comando en una nueva capa encima de la imagen base
`ENV <key> <value>`	Esta instrucción configura las diferentes variables de entorno
`ADD <origen> <destino>`	Esta instrucción copia los archivos o directorios en la ubicación especificada del sistema de archivos del contenedor. Permite también la extracción de ficheros y soporte de URLs remotas, por lo cual su funcionamiento es más complejo que el de `COPY`
`COPY <origen> <destino>`	Al igual que `ADD`, añade los archivos en la ruta especificada, pero en este caso solo copiará los mismos
`EXPOSE <port>`	Esta instrucción especifica a Docker los puertos en los que deberá escuchar el contenedor pero además será necesario mapear los mismos utilizando la opción `-p` de `docker run`.

Comando	Descripción
`CMD ["ejecutable", "param1", "paramn"]`	Esta instrucción provee de valores por defecto al contenedor. Cabe destacar que solo tendrá validez la última aparición del comando.
`ENTRYPOINT ["ejecutable", "param1", "paramn"`	Esta instrucción permite pasar cualquier argumento al punto de entrada, pero anulará cualquier elemento especificado con `CMD`
`VOLUME /path`	Crea un punto de montaje con un nombre especificado y lo marca con un volumen montado externamente desde el host y otro contenedor.
`USER <username>`	Configura el nombre de usuario a utilizar cuando se lanza un contenedor
`WORKDIR <path>`	Especifica el directorio de trabajo de cualquier otra instrucción
`HEALTHCHECK <command>`	Permite añadir un **healthcheck** al contenedor con el fin de comprobar periódicamente el estado del mismo.

> La verificación del estado de un contenedor, o **healthcheck**, nos permitirá detectar problemas de servicio cuando el proceso está en ejecución.
>
> También se podrá hacer a través del fichero Docker Compose, el cual estudiaremos más adelante

Tras completar la creación de nuestro fichero Dockerfile y para que todas las instrucciones existentes en el fichero puedan ser leídas de manera sucesiva y pueda, así, crearse la imagen tendremos que usar el comando «**`docker build`**».

```
docker build -t <image-name> <dockerfile-path>
```

Véase una explicación más detallada en el siguiente enlace:

https://docs.docker.com/engine/reference/builder/

TÉRMINOS:

tag: etiqueta que se utiliza para identificar las versiones de las imágenes utilizadas, siendo este tag el alias de su ID. [12]

Variables de entorno: conjuntos de valores dinámicos con nombre, almacenados en la memoria, que afectan a los programas o procesos que se ejecutan en un servidor, pudiendo estos ser utilizados por varios procesos que funcionan de manera simultánea. En otras palabras, son variables con un nombre y un valor asociado, que permiten personalizar el funcionamiento del sistema y el comportamiento de sus aplicaciones. [13] [14] [15]

Puerto (port) :punto de conexión entre redes en la que los paquetes de información se dirigen hacia la dirección de destino final. También se define como el medio por el cual un programa cliente se comunica con un programa específico en una computadora conectada a una red. [16][17]

Mapear: asignar de puertos lo que permite a una máquina el establecer diversas conexiones con máquinas diferentes de manera simultánea.

Volumen: fichero que se monta directamente en un contenedor, de tal forma que si el contenedor es borrado, la información contenida en el, persiste.[18]

Comandos útiles para la gestión de contenedores

Una vez terminada la construcción de la imagen, tendremos que hacer, ahora, uso de otros comandos que nos permitirán crear nuestro contenedor y gestionar el mismo. En concreto:

- Para **crear un contenedor** a partir de una imagen:

```
docker run -it --name <container-name> -p <host-
port>:<container-port> -d <image-name>
```

- Para ver los **contenedores existentes** que se encuentren levantados:

```
docker ps
```

- Para **ver todos los contenedores**, independientemente de su estado:

```
docker ps -a
```

- Para **acceder al contenedor** o ejecutar comandos en el mismo:

```
docker exec -it <container-name|id> <command>
```

- Por ejemplo, para acceder por terminal, o línea de comandos, podríamos utilizar el comando:

```
docker exec -it <container-name> /bin/bash
```

`/bin/bash` indica que los scripts o comandos ejecutados serán interpretados con Bash, el *shell* (o intérprete de comandos) de Linux. Lo veremos en más ocasiones más adelante. [19]

- Para comprobar el **Healthcheck**:

```
docker inspect --format='{{json .State.Health}}' <container-
name>
```

- Para ver los logs:

```
docker logs -ft <container-name>
```

> Los logs, o registros, nos ayudan a solucionar posibles problemas en el uso de Docker, mostrándonos la información de lo que está sucediendo en cualquier proceso a tiempo real.

Primer Dockerfile

Tras conocer los comandos principales para trabajar con el Dockerfile, crearemos ahora, a modo de práctica, nuestro primero fichero Dockerfile básico, al cual le instalaremos la herramienta «htop», que servirá para monitorizar nuestro sistema a modo de monitor de procesos interactivo. El código del fichero quedaría de la siguiente manera:

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get -y install htop

CMD ["bash"]
```

Al final del fichero anterior añadimos un comando CMD (command) con el proceso de [«bash»] para mantener vivo dicho contenedor. Esto se debe a que es necesario que al menos un proceso se esté ejecutando en el mismo para mantenerlo levantado; en el momento en que dicho proceso principal muere, el contenedor moriría con él.

Buenas prácticas en el uso de Dockerfile

A continuación ofrecemos una lista de buenas prácticas relacionadas con el uso del fichero Dockerfile, las cuales nos permitirán hacer un uso más óptimo y eficiente del mismo:

1. Al crear una imagen, Docker debe preparar el contexto (*context*) como primera acción. El contexto predeterminado contiene todos los archivos del directorio Dockerfile, pero en ocasiones no se desea incluir todos los archivos existentes. Para ello, se recomienda el uso de un archivo .«dockerignore», en el que se indicarán los archivos que se deben excluir del context de Dockerfile.
2. Cada contenedor debe hacer una sola cosa.
3. En la práctica, se podrían indicar múltiples procesos dentro de un mismo contenedor, pero esto solo provocaría consecuencias indeseables para el mismo, por ejemplo:
    - Tiempos de compilación muy largos
    - Imágenes muy pesadas
    - Grandes ficheros de log
    - Escalado horizontal innecesario
    - Problemas con **procesos zombies**

    > Procesos zombies: aquellos que permanecen en la tabla de procesos incluso después de que hayan completado la ejecución. [Los estudiaremos en profundidad más adelante]
4. Fusionar varios comandos `RUN` en uno solo, reduciendo el número de capas del contenedor [*véase* Tabla 1].
5. Especificar una etiqueta de versión, evitando con ello el uso por defecto de la etiqueta `«latest»` en la imagen base y evitando así problemas con cambios de versiones no deseadas.

6. Eliminar archivos innecesarios después de cada paso de la instrucción `RUN`
7. Utilizar una imagen base adecuada o preparada para la aplicación que realmente se desea utilizar.
8. Configurar directorios por defecto con `WORKDIR`
9. Ejecutar comandos específicos mediante las instrucciones `CMD` o `ENTRYPOINT`
10. Utilizar la instrucción `COPY` en lugar de `ADD`, ya que esta última consume más recursos a la hora de crear la imagen.
11. Optimizar las instrucciones `COPY` y `RUN` evitando repetir las mismas en varias ocasiones
12. Especificar variables de entorno, puertos y volúmenes predeterminados
13. Añadir metadatos a la imagen mediante la instrucción `LABEL`
14. Añadir la instrucción `HEALTHCHECK` para conocer el estado de los contenedores

Véase el siguiente link para profundizar sobre estas recomendaciones:

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Ejercicio práctico

Para el aprendizaje de la creación de imágenes personalizadas con Dockerfile, se propone como práctica el montaje de 3 contenedores diferentes con los servidores web **Nginx** y **Apache Tomcat,** y el gestor de bases de datos **MySQL**, creando con ello un entorno completo.

Para ello se han creado diferentes ficheros que permitirán crear los entornos indicados. Dichos recursos pueden descargarse desde el siguiente enlace:

https://github.com/shokone/Vagrant-Docker/raw/master/resources.tar.gz

Los 3 contenedores indicados quedarán de la siguiente manera:

Contenedor Nginx

```
FROM nginx:1.17
LABEL MAINTAINER "imartinez@example.com"
RUN apt-get update -y \
     && apt-get upgrade -y \
     && apt-get install curl -y \
     && rm /etc/nginx/conf.d/default.conf

COPY html /usr/share/nginx/html
COPY conf /etc/nginx/conf.d

ENV APP_PORT=80
EXPOSE $APP_PORT

HEALTHCHECK CMD curl -s --fail http://localhost:$APP_PORT || exit
```

Contenedor Tomcat

```
FROM tomcat:8.5.50-jdk8
LABEL MAINTAINER "imartinez@example.com"
ENV JDBC_DDBB=example_db \
        JDBC_URL=jdbc:mysql://10.100.1.200:3306/example_db?
connectTimeout=0&amp;socketTimeout=0&amp;autoReconnect=true \
        JDBC_USER=db_user \
        JDBC_PASS=db_password \
        APP_PORT=8080
COPY code/webapps /usr/local/tomcat/webapps
EXPOSE $APP_PORT
HEALTHCHECK CMD curl -s --fail http://localhost:$APP_PORT || exit
1

CMD ["catalina.sh", "run"]
```

Contenedor MySQL

```
FROM mysql:5.6

ENV MYSQL_ROOT_PASSWORD=mysqlpassword \
        MYSQL_DATABASE=example_db \
        MYSQL_USER=db_user \
        MYSQL_PASSWORD=db_password \
        APP_PORT=3306

COPY data/mysql-init.sql /docker-entrypoint-initdb.d/init.sql

EXPOSE $APP_PORT

CMD ["mysqld"]
```

CAPÍTULO 4

Conociendo Docker

Repositorios de imágenes públicos y privados

El sitio web de DockerHub[3] (servicio proporcionado por Docker) permite la creación de repositorios de imágenes públicos y privados con un simple registro. De esta manera podremos subir imágenes o buscar y descargar y usar imágenes oficiales ya creadas de multitud de proyectos.

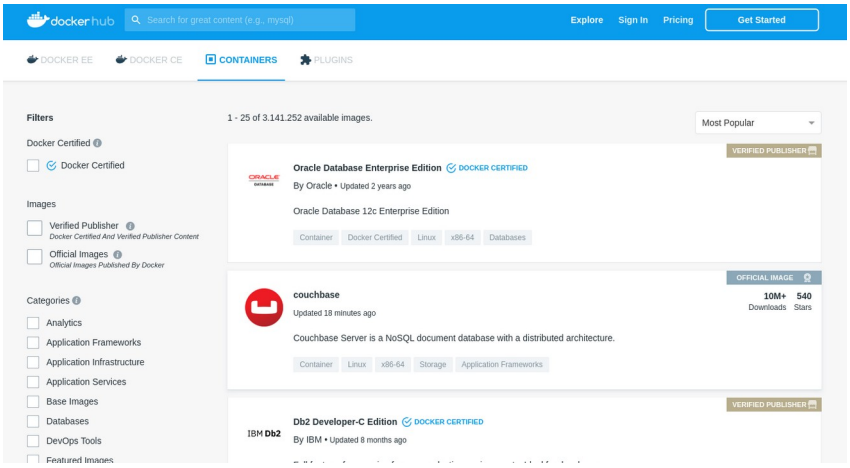

Figura 6: Captura de pantalla del repositorio de DockerHub

Sin necesidad de pagar un plan, de manera gratuita, DockerHub nos permitirá:

1. Acceder a todos los repositorios públicos
2. Crear repositorios públicos de forma ilimitada
3. Crear UN repositorio privado repositorio privado

Para acceder a repositorios privados o crear más de uno de ellos, será necesario pagar un plan.

3 https://hub.docker.com/search/?type=image

Seguridad en el uso de imágenes públicas

Gran parte del éxito de Docker se basa en la facilidad de distribuir y compartir imágenes. Probablemente existe un contenedor publicado en Docker Hub para cualquier cosa que podamos necesitar, y usarlo es tan sencillo como ejecutar un `docker run` o construir nuestro propio contenedor con la instrucción `FROM`.

Pero esta facilidad de uso nos debe hacer cautos. Pueden existir imágenes que introduzcan problemas de seguridad, ya sea intencionadamente (imágenes maliciosas creadas para minar monedas, filtrar información, etc.), o simplemente imágenes creadas con malas prácticas, bibliotecas con vulnerabilidades, etc. que podrían comprometer nuestros sistemas si decidimos usarlas.

Por ello, siempre debemos asegurarnos de que las imágenes utilizadas vengan de proveedores oficiales y estén verificadas y actualizadas, además de realizar un análisis de vulnerabilidades para detectar posibles problemas.

Entre Mayo de 2017 y Mayo de 2018 se dieron casos de imágenes maliciosas subidas a Docker Hub (llamadas Docker123321). En total fueron descargadas más de 5 millones de imágenes maliciosas antes de que fueran eliminadas el 10 de Mayo de 2018. En este enlace de Artechnica.com[4] se puede leer la noticia completa .

4 https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/

CAPÍTULO 5

Docker Internals

Docker internals

Docker es básicamente un conjunto de utilidades que incluye un *runtime* de contenedores (permite la ejecución de contenedores) y que también incorpora comandos para la construcción, empaquetado y distribución de los mismos, con

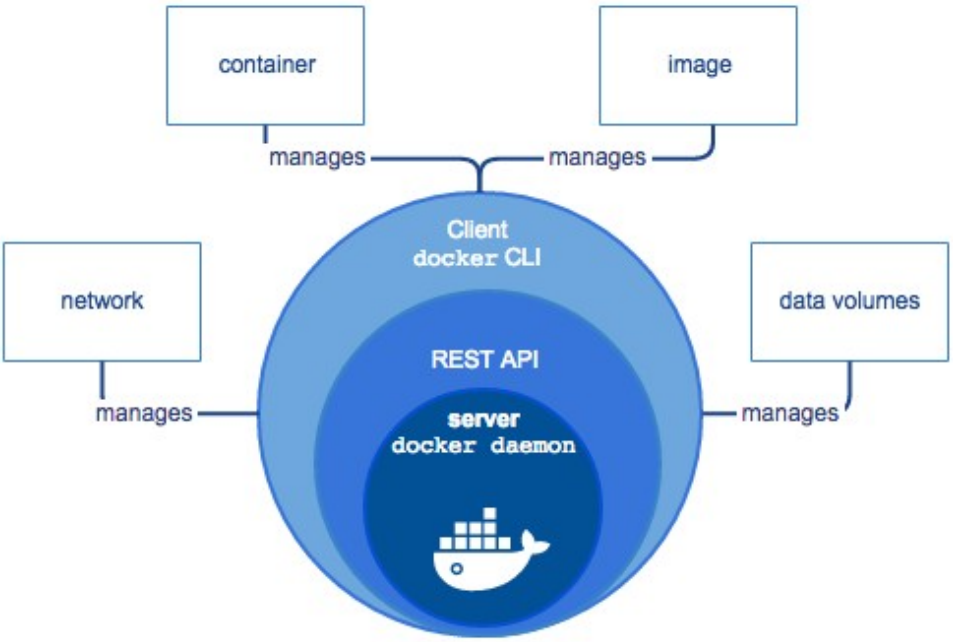

Figura 7: Estructura monolítica original de Docker

una arquitectura cliente/servidor.

Originalmente era una aplicación monolítica (el *daemon* dockerd), junto a una aplicación cliente de línea de comandos CLI (Docker). El daemon proveía casi toda la lógica para construir contenedores, gestionar las imágenes y ejecutarlos, y una API. Se podía, o bien usar directamente la API de dockerd, o bien mandar los comandos a través del cliente de línea de comandos, que se conecta a la API

de dockerd.

Runtime en el sentido de entorno de ejecución, hace referencia a aquella máquina virtual que porporciona servicios a otro programa que se está ejecutando. [8]

daemon, en computación, es un tipo de programa que se ejecuta de forma no interactiva (como un proceso en segundo plano) , en vez de estar bajo el control del usuario. **Dockerd** es el daemon de Docker, así como el comando utilizado para ejecutar el mismo, a través del cual podremos administrar los objetos de Docker (imágenes, contenedores, redes, volúmenes, etc.). Además, podrá conectarse con otros daemons para administrar los servicios de Docker. [] [20]

API es la interfaz de programación de aplicaciones, que permitirá que Docker interactúe con el daemon dockerd.

Arquitectura de Docker

Aunque el diagrama anterior (Figura 7) sigue siendo válido como una visión de alto nivel de Docker, posteriormente esta estructura monolítica fue separada en tres partes (Figura 8):

- *docker-containerd:* la parte alto nivel del runtime de ejecución de contenedores. Se encarga también de la gestión de imágenes, así como de la ejecución
- **dockerd**: el daemon que provee funcionalidad, a través de la API, para construir imágenes. La lógica de construcción, de hecho, simplemente interpreta los comandos del fichero Dockerfile, ejecuta los comandos correspondientes dentro de un contenedor usando containerd, y guarda el sistema de archivos resultante del contenedor como una imagen.
- *docker-runc:* la parte bajo nivel del runtime de ejecución.

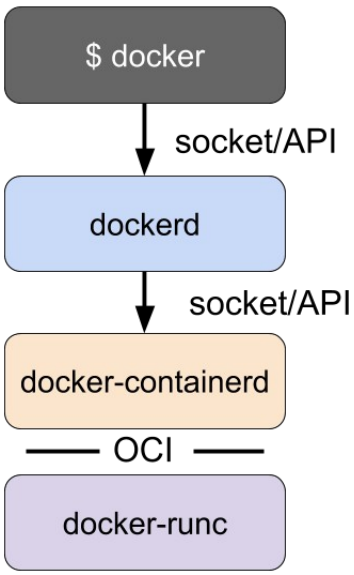

Figura 8: Evolución de la estructura de Docker

> *docker-containerd* y *docker-runc* son versiones empaquetadas por Docker de los correspondientes containerd y runc originales.

Si bien, con esta estructura (Figura 8), se podría hablar directamente con la API de Docker (existen kits de desarrollo de software SDKs y clientes de la API para distintos lenguajes), lo habitual suele ser, tal y como se muestra en el esquema de la Figura 9, utilizar el **cliente** de línea de comandos Docker, que se encarga de comunicarse con dockerd (el **Docker daemon**), trasladar la solicitud y mostrar al

usuario el resultado en un formato más apropiado.

La *API* de Docker es una API REST. La comunicación se puede realizar usando un *socket* de dominio UNIX o un interfaz de red, y el cliente y el daemon pueden estar en distintas máquinas.

> Socket (o enchufe): objeto de software que actúa como un punto final, estableciendo un enlace en un programa entre el lado del servidor y el del cliente. En UNIX, un socket es un punto final para la comunicación entre procesos dentro del sistema operativo.[21]
>
> API REST: Tipo de API que permite una mejor comunicación entre aplicaciones ya que establece ciertas restricciones para que esta comunicación se realice de forma efectiva.[22]

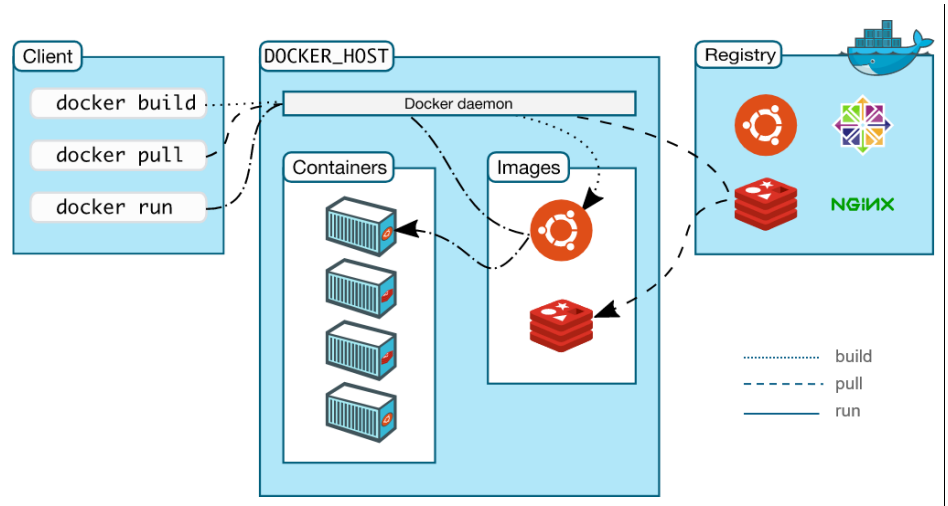

Figura 9: Arquitectura de Docker: cliente, daemon y registro

A continuación estudiaremos estos tres componentes de la arquitectura de Docker (**cliente, daemon y registro**) por separado y en mayor profundidad.

El daemon Docker

Es el componente que se encarga de escuchar las peticiones a la API y gestionar los objetos Docker como: imágenes, contenedores, redes y volúmenes. **Dockerd** (**o daemon de Docker**), a su vez, como hemos comentado en la introducción, se comunica con otros daemons, como containerd, para realizar algunas funciones.

El puerto en el que escucha dockerd se puede cambiar con la opción `-H` al lanzar el daemon. El valor por defecto, si no se especifica, es:

```
unix:///var/run/docker.sock
```

Esto hace que el puerto escuche solo localmente en un Unix Socket en `/var/run/docker.sock`. Cambiando los permisos de acceso a `/var/run/docker.sock` se puede permitir que otros grupos o usuarios puedan conectarse a Docker.

También se puede especificar una opción como: `tcp://0.0.0.0:2375` para escuchar vía **TCP** en todos los interfaces en el puerto 2375, o especificar una IP concreta. Sin embargo no es recomendable por cuestiones de seguridad, ya que permitir acceso al **socket de Docker** hace trivial conseguir acceso root (superusuario) a la máquina (el daemon se ejecuta como root). En este caso es recomendable al menos habilitar ***-tls*** (por defecto puerto 2376 para TLS) y poner un servidor WebProxy delante para controlar el acceso.

> TCP: El protocolo TCP (Protocolo de Control de Transmisión) es, junto con el IP (Internet Protocol) uno de los protocolos más importantes y más usados de la familia de protocolos de red en los que se basa Internet. Se caracteriza por garantizar que los datos serán entregados en su destino sin errores y en el mismo orden en el que salieron. [23] [24]
>
> El socket de Docker es el socket utilizado para la comunicación con el docker daemon principal. La CLI de Docker utilizará este socket por defecto para ejecutar los comandos. [25]
>
> TSL**:** Evolución del protocolo SSL. Garantiza el intercambio de datos entre usuario y servidor de forma encriptada y en un entorno privado..

El cliente Docker

Es el principal modo de interactuar con Docker. Al ejecutar comandos como `docker run`, el cliente envía peticiones a API REST del daemon, que los lleva a cabo. El mismo cliente Docker puede comunicarse con varios daemons.

Se puede especificar la dirección del EndPoint de la API del dockerd mediante la variable de entorno `DOCKER_HOST` o bien el parámetro `-H` o `--host` en el cliente Docker:

```
DOCKER_HOST=tcp://X.X.X.X:2375
```

Registros de Docker

El registro es el lugar donde se almacenan las imágenes. Docker Hub es un registro público y gratuito para imágenes públicas, y con versiones de pago para registros privados (véase página 47). En el registro público, cualquiera puede hacer `push` de sus imágenes, o `pull` de las imágenes creadas por otros usuarios. Es decir, cuando se ejecuta `docker pull` o `docker run`, las imágenes requeridas se descargan del registro. Cuando se hace `docker push`, la imagen se sube al registro.

Docker viene configurado por defecto para buscar las imágenes de Docker Hub (`docker.io/`) a no ser que se indique explícitamente otro registry (por ejemplo `gcr.io/myimage:latest` para el Google Container Registry).

También es posible, como veremos, alojar nuestro propio registro. Podemos usar la misma implementación *vanilla* del Docker registry que usa Docker Hub, o usar otras más avanzadas como el registro *open source* (de código abierto) Harbor (VMWare), Artifactory (JFrog), Nexus (Sonatype), etc.

Docker también ofrece una *versión enterprise* de su registro que se llama «Docker Trusted Registry (DTR)», y existen otras opciones de pago (o gratuitas, según modalidad) como las siguientes:

- Google Container Registry (GCR)
- Amazon Elastic Container Registry (ECR)
- Azure Container Registry (ACR)
- Quay (Redhat, versión On-Prem y versión cloud)
- Gitlab Container Registry
- Github Packages

Se estudiarán en profundidad todos estos registros y sus posibilidades a lo largo del Registros de Docker de esta guía.

Tipos de objetos en Docker

Tras estudiar los tres componentes básicos que conforman la arquitectura de Docker, estudiaremos ahora los tipos de objetos existentes en ella.

Imágenes

Una **imagen Docker** es una plantilla con las instrucciones necesarias para crear un contenedor Docker. Se pueden crear múltiples contenedores a partir de una misma imagen, y parametrizarse mediante variables de entorno, montando distintos volúmenes, etc. para adaptar su funcionamiento.

Las imágenes están compuestas de ***layers*** **(capas)**, que son el resultado de la ejecución de cada uno de los comandos del Dockerfile (cada capa representa una instrucción). Como ya se ha comentado, dockerd, en la práctica, lo que hace es interpretar el Dockerfile y ejecutar cada paso utilizando containerd, generando así un contenedor intermedio. Cada una de estas capas, como veremos más adelante, contiene únicamente los cambios respecto a la capa anterior.

Es habitual que las imágenes no se creen desde cero (`FROM scratch`), sino que estén basadas en otra imagen, creada por terceras personas y publicada en un registro, a la que se añaden capas de personalización adicionales. Por ejemplo, puedes partir de un contenedor construido sobre los sistemas operativos Debian, Alpine o Ubuntu y añadir los paquetes y la configuración de un servidor Apache específicos para nuestra aplicación.

Al estar las imágenes compuestas por capas, el proceso de construcción es más rápido si se parte de imágenes ya pre-construidas, haciendo así un uso intensivo de la caché para reconstruir únicamente las capas cuyos comandos hayan cambiado. Si Docker detecta que ejecutamos el mismo comando, reutilizará la misma capa de la caché (a no ser que usemos la opción `--no-cache`).

Cuando se construyen y se publican en el registro, a las imágenes se les añade un ***tag,*** o etiqueta, para identificar así su versión o variante. Es habitual utilizar la nomenclatura de **repositorio** para nombrar a una imagen independientemente de su etiqueta. Por ejemplo, en el caso del controlador NGINX Ingress Controller, se denominaría el repositorio `nginx/nginx-ingress`, que contiene todas las imágenes `nginx/nginx-ingress:some-tag`

Contenedores

Como ya se ha explicado, un **contenedor** es una instancia ejecutable de una imagen. De esta manera, se pueden realizar distintas acciones sobre los contenedores, por ejemplo: crear un contenedor (pero no arrancarlo), iniciar (*start*), detener (*stop*), pausar y resumir, borrar, conectarse a un contenedor en ejecución, crear una nueva imagen basada en el estado de un contenedor, etc. Todo esto se realizará a través de la API o usando el cliente CLI (interfaz de línea de comandos).

Cada contenedor se ejecuta en un entorno aislado, con sus propias variables de entorno, sus volúmenes montados, sus propios interfaces de red y las opciones de configuración que se usan al crearlo o arrancarlo. Por defecto, está aislado de otros contenedores y de la máquina host, aunque el nivel de aislamiento se puede controlar al crearlo.

Al borrar un contenedor, cualquier cambio en su estado es eliminado, a no ser que se haya almacenado en almacenamiento persistente (como un volumen).

Servicios

Docker Compose es una herramienta de Docker que nos permitirá definir y ejecutar aplicaciones Docker de múltiples contenedores, simplificando así el uso de Docker. Veremos este concepto en profundidad en el Capítulo Docker Compose de esta guía.

Por su parte, la herramienta orquestadora de contenedores **Docker Swarm**, nos permitirá la planificación de estos en el caso de que se esté utilizando un *cluster* de servidores (grupo de servidores que funcionan como si de uno solo se tratase), a través de una API.

Veamos cómo se usa el concepto de **servicio** aplicado a estas dos herramientas:

- En **Docker Compose**, un servicio define todos los contenedores que componen una aplicación multi-contenedor, que son gestionados conjuntamente, y que, normalmente, se conectan a una misma red Docker para poder comunicarse entre sí. Por ejemplo, es habitual levantar un contenedor de base de datos además del contenedor que lo utiliza.
- En **Docker Swarm**, los servicios se usan para escalar contenedores en múltiples Docker daemons (distintas máquinas), pero trabajando de forma conjunta. Cada miembro de un *swarm* es un Docker daemon, que se comunica con los otros. De este modo se puede lanzar un servicio indicando un número de réplicas de un contenedor, siendo la apariencia desde fuera de un único punto de entrada. Un balanceador se encargará de enviar las peticiones a las distintas instancias del contenedor en distintas máquinas.

Open Container Initiative (OCI)

Open Container Initiative (OCI) es un proyecto creado en 2015 por Docker para el diseño de **estándares en la gestión de contenedores** Linux. Actualmente forman parte de este proyecto importantes compañías como Google y Redhat, que compró CoreOS y por tanto rkt (CoreOS Rocket). [26]

Para acceder a la página web oficial de OCI y profundizar en los estándares entrar en:

https://www.opencontainers.org/

Figura 10: Logo Open Container Iniciative (OCI)

La especificación de OCI se compone de dos partes: la especificación de imagen (i*mage-spec*), que indica el formato de empaquetado de una imagen de contenedor, y la especificación de ejecución (*runtime-spec*), que define un *filesystem bundle* (formato de desempaquetado en el disco) y cómo ejecutarlo.

La **especificación de imagen** (*image-spec)* define el formato de una imagen OCI, que se compone de:

- Un *manifest,* esto es, un archivo que contiene los metadatos y dependencias de la imagen, incluyendo la identidad (*content-addressable*) de uno o más archivos, con la serialización del contenido de las capas que se desempaquetan para generar el sistema de archivos final del contenedor.
- La configuración de la imagen, como argumentos, variables de entorno, etc.
- La serialización de los sistemas de archivos con el contenido de las capas.

La **especificación de ejecución** (*runtime-spec)* define la configuración, el entorno de ejecución y el ciclo de vida de un contenedor, para conseguir así un entorno y comportamiento consistente entre distintos runtimes, así como acciones comúnmente realizadas para controlar el ciclo de vida del contenedor.

Containerd

Cuando se rompió la arquitectura monolítica de dockerd, se creó ***containerd*** para gestionar la funcionalidad de contenedores, imágenes, `pull` y `push` (descargar y subir imágenes, respectivamente) desde el repositorio, etc. En esta entrada del Blog de Docker[5] se explican las ventajas que ha traído containerd a Docker

Figura 11: Logo de containerd

Containerd implementa las especificaciones OCI para gestionar y ejecutar imágenes, estructurándose tal y como se muestra en la siguiente imagen [Figura 12]:

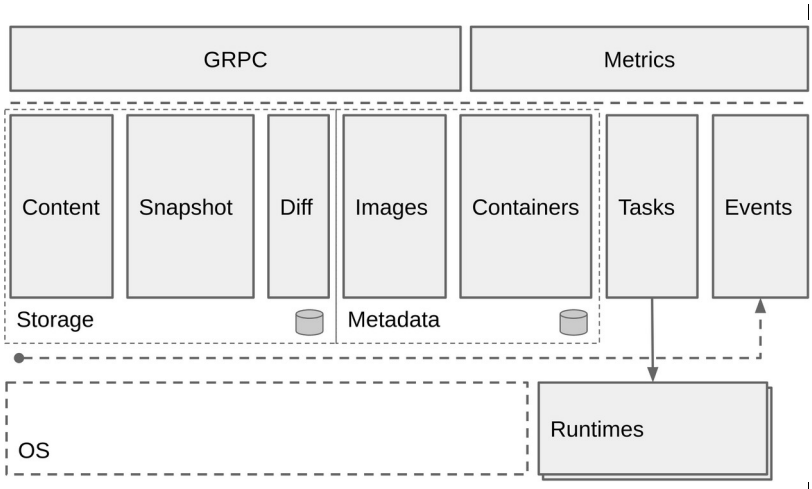

Figura 12: Estructura del daemon containerd

Para acceder a la página web oficial de containerd y profundizar en los estándares entrar en:

https://containerd.io/

5 https://www.docker.com/blog/docker-containerd-integration/

En containerd, se usa la herramienta runc como el runtime por defecto, pero puede usar cualquier otro runtime compatible con la especificación OCI.

Se puede usar el cliente de línea de comando **ctr** para interactuar con **containerd** como se muestra a continuación:

```
$ sudo ctr images pull docker.io/library/redis:latest
ctr: failed to resolve reference "docker.io/library/redis": object required
docker.io/library/redis:latest:                                   resolved
|++++++++++++++++++++++++++++++++++++++|
index-sha256:90d44d431229683cadd75274e6fcb22c3e0396d149a8f8b7da9925021ee75c30:
done
|++++++++++++++++++++++++++++++++++++++|
manifest-
sha256:e4b315ad03a1d1d9ff0c111e648a1a91066c09ead8352d3d6a48fa971a82922c: done
|++++++++++++++++++++++++++++++++++++++|
layer-sha256:727f8da63ac248054cb7dda635ee16da76e553ec99be565a54180c83d04025a8:
done           |++++++++++++++++++++++++++++++++++++++|
config-sha256:9b188f5fb1e6e1c7b10045585cb386892b2b4e1d31d62e3688c6fa8bf9fd32b5:
done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:8ec398bc03560e0fa56440e96da307cdf0b1ad153f459b52bca53ae7ddb8236d:
done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:da01136793fac089b2ff13c2bf3c9d5d5550420fbd9981e08198fd251a0ab7b4:
done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:cf1486a2c0b86ddb45238e86c6bf9666c20113f7878e4cd4fa175fd74ac5d5b7:
done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:a44f7da98d9e65b723ee913a9e6758db120a43fcce564b3dcf61cb9eb2823dad:
done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:c677fde73875fc4c1e38ccdc791fe06380be0468fac220358f38c910e336266e:
done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 6.9 s
total: 33.9 M (4.9 MiB/s)
unpacking linux/amd64
sha256:90d44d431229683cadd75274e6fcb22c3e0396d149a8f8b7da9925021ee75c30...
done

$ sudo ctr images list

REF                            TYPE
DIGEST                                                                  SIZE
PLATFORMS
LABELS
docker.io/library/redis:latest
application/vnd.docker.distribution.manifest.list.v2+json
sha256:90d44d431229683cadd75274e6fcb22c3e0396d149a8f8b7da9925021ee75c30 34.1
MiB linux/386,linux/amd64,linux/arm/v5,linux/arm/v7,linux/arm64/v8,linux/
ppc64le,linux/s390x -

$ sudo ctr container create docker.io/library/redis:latest redis

$ sudo ctr container list
```

```
CONTAINER    IMAGE                                RUNTIME
redis        docker.io/library/redis:latest    io.containerd.runtime.v1.linux

$ sudo ctr container delete redis
```

Las capas y datos de las imágenes se guardan en `/var/lib/containerd`

A priori, puede parecer que la funcionalidad de `ctr` para la interacción con containerd es parecida a la del comando `docker`, pero sin embargo, containerd está centrado tanto en la ejecución de contenedores (runtime) como en el uso de operaciones (como ejecutar contenedores en un servidor), y no incluye ningún mecanismo para la construcción de imágenes. Docker, por otra parte, está más orientado a desarrolladores y al usuario final.

Containerd tiene una API para el *framework* gRPC (también existe la variante ttrpc para entornos con poca memoria), y existe un plugin para implementar Kubernetes CRI (Container Runtime Interface), permitiendo usar containerd en la plataforma Kubernetes. Obsérvese esto en la siguiente imagen [Figura 13], en la que se describe esquemáticamente la arquitectura del daemon containerd.

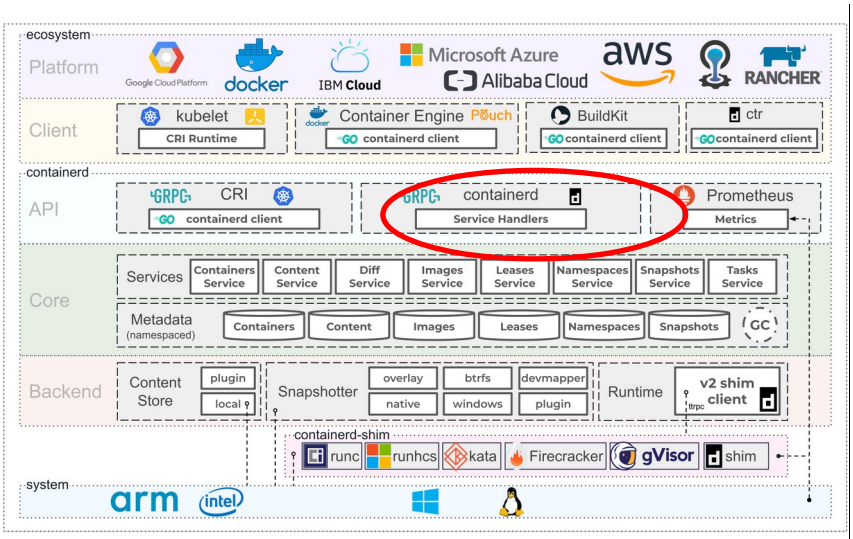

Figura 13: Arquitectura de containerd

Existen, además, clientes para la API de containerd, como por ejemplo el paquete containerd de Go, al que podemos acceder a través de este enlace:

https://godoc.org/github.com/containerd/containerd

containerd se encarga únicamente de la gestión de **contenedores**. El docker daemon (engine) se sigue encargando de la gestión de volúmenes, red, etc.

Containerd-shim

Cuando se crea un nuevo contenedor, containerd lanza un nuevo proceso que es el padre del contenedor, llamado «*containerd-shim*» (literalmente, contenedor de cuña o arandela). Este nuevo proceso «*shim*», permite las siguientes cuestiones:

- El shim permite contenedores *daemonless*, es decir, que no dependen de containerd. El proceso shim se convierte en el padre del contenedor.

- El shim permite al runtime, por ejemplo runC, que termine su proceso tras lanzar el contenedor (y hereda el proceso hijo del contenedor). De esta manera, no es necesario mantener en ejecución del proceso runc. Si lanzas un contenedor MySQL, en la lista de procesos sólo verás mysql y el containerd-shim.
- Mantiene STDIO y otros descriptores de archivo (FDs) abiertos aunque containerd o *docker* mueran, de forma que el contenedor puede seguir ejecutándose. Esto es un gran cambio, y permite, por ejemplo, actualizar Docker sin parar a todos los contenedores, como ocurría en versiones antiguas.
- Permite reportar el *exit status* del contenedor (para indicar si el software funcionó con éxito) sin tener que hacer un *wait* desde el containerd o *docker.*

Runc

Runc es el runtime o sistema de ejecución de contenedores usado por Docker. Originalmente formaba parte del sistema monolítico de Docker, y posteriormente fue extraído como una biblioteca y herramienta independiente, como parte de la OCI.

Por lo tanto, runc implementa la *runtime-spec* de OCI, y ejecuta contenedores en formato OCI, que es básicamente un archivo tipo *config.json* (véase a continuación) con la especificación, y el sistema de archivos del contenedor que se monta como *rootfs* (sistema de archivos raíz).

> Recordemos que el *runtime-spec* es la especificación de ejecución, uno de los dos componentes de la especificación de OCI [véase Open Container Initiative (OCI)]

> JSON (JavaScript Object Notation) es un formato para el almacenamiento e intercambio de datos, basado en texto escrito con notación de objetos JavaScript. [27]

Así, se puede instalar el proyecto y ejecutar un contenedor de la siguiente manera:

```
$ mkdir rootfs
$ docker export $(docker create busybox) | tar -xf - -C rootfs

$ runc spec
```

Crea un fichero config.json:

```
{
     "ociVersion": "1.0.0",
     "process": {
           "terminal": true,
           "user": {
                 "uid": 0,
                 "gid": 0
           },
```

```
            "args": [
                  "sh"
            ],
            "env": [


PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:usr/bin:/sbin:/bin",
                  "TERM=xterm"
            ],
            "cwd": "/",
            "capabilities": {
...
```

```
$ sudo runc run mycontainerid
/ # echo "Hello from in a container"
Hello from in a container
```

Para ampliar la información acerca de runc, pueden consultarse los siguientes enlaces:

https://www.docker.com/blog/runc/

https://github.com/opencontainers/runc

- Breaking out of Docker via runC – Explaining CVE-2019-5736

La vulnerabilidad CVE-2019-5736 permite que un contenedor malintencionado (con una mínima interacción del usuario) sobreescriba el binario de runc del host y, por lo tanto, obtenga la ejecución del código a nivel raíz sobre el host. El nivel de interacción del usuario es poder ejecutar cualquier comando como root dentro de un contenedor en los contextos de crear un nuevo contenedor a partir de una imagen controlada por el atacante u obtener una shell en un contenedor sobre el cual tenga permisos de escritura el atacante.

A continuación se muestra un enlace con la explicación y una POC de la

vulnerabilidad mencionada.

https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/

Libcontainer

Docker en sus orígenes utilizaba la interfaz de usuario LXC como “motor” para la creación de contenedores. Es decir, LXC era la parte encargada de gestionar y crear los *namespaces*, *cgroups*, etc. para preparar el entorno de ejecución del contenedor. [Para profundizar sobre el concepto de la interfaz LXC, puede entrar aquí[6]].

En marzo de 2014, desde la versión 0.9 de Docker, la compañía decide escribir su propia biblioteca, conocida como ***libcontainer***, eliminando así esta dependencia, que pasará a formar parte de runc. De esta manera, tal y como se muestra en el siguiente esquema [Figura 14], Docker utilizará esta biblioteca libcontainer como interfaz para acceder a las diferentes capacidades de virtualización de Linux (cgroups, namespace, etc.) [6].

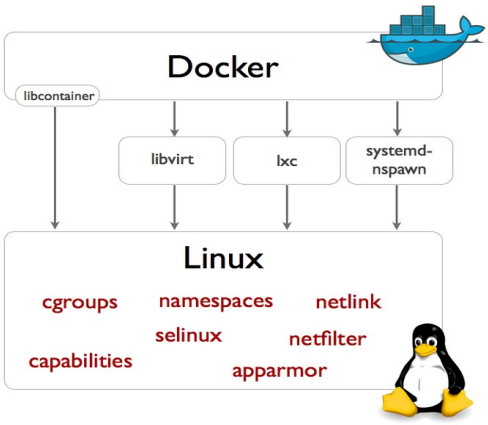

Figura 14: Uso de libcontainer para acceder a facilidades Linux

Gracias a esto, se puede crear una capa de abstracción, escrita en C/C++, preparada para soportar otras tecnologías de virtualización, como contenedores

6 https://github.com/lxc/lxc

en Windows, contenedores en otras arquitecturas, etc.

Por tanto, podríamos considerar libcontainer como una «*bifucación fork*», o reescritura de LXC, que forma parte del entorno de ejecución runc.

Para ampliar la información acerca de libcontainer, pueden consultarse los siguientes enlaces:

https://github.com/opencontainers/runc/tree/master/libcontainer

http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/

BuildKit

Aunque no está dentro del paraguas ni de la especificación de OCI, cabe mencionar que Docker está trabajando en un reemplazo del actual motor de construcción de Docker (que lleva varios años sin apenas cambios significativos). Este reemplazo recibe el nombre de **BuildKit**, y permitirá independizar la parte de construcción de contenedores del daemon Docker. Hablaremos más adelante sobre sistemas alternativos de construcción de contenedores, incluyendo BuildKit.

Para ampliar la información acerca de Builtkit, pueden consultarse el siguiente enlace:

https://github.com/moby/buildkit

Namespaces

Un espacio de nombres, o ***namespace***, es una tecnología del kernel de Linux que provee espacios de trabajo separados y una capa de aislamiento, haciendo que cada contenedor solo pueda ver los recursos pertenecientes a su namespace. De esta forma, Docker (runc), al crear un contenedor, crea también un conjunto de namespaces para dicho contenedor.

Docker utiliza los siguientes tipos de namespaces, según su función:

- **pid (Process ID)**: Aislamiento de procesos
    - Sólo se ven procesos en el mismo namespace.
    - Sigue su propia numeración, comenzando en PID 1.
    - Si el PID 1 muere, se elimina todo el namespace.
    - Como los namespaces pueden estar anidados, un proceso puede acabar teniendo varios PIDs, uno por cada namespace.
- **net:** interfaces de red (incluyendo la interfaz *loopback*), tablas de rutas, reglas de *iptables*, sockets (comandos ss, netstat).
    - Cada interfaz de red está presente exactamente en un namespace, y se puede mover entre namespaces:

      ip link set dev eth0 netns PID
    - Usos típicos:
        - Pares de dispositivos *veth (*es decir, *virtual Ethernet Devices)* en un namespace, actuando como un cable cable *crossover* entre ellos. Se pone uno en el container, y otro en el host, y hace de *switch* virtual, permitiendo que estos se comuniquen.
        - Poner el nombre de la interfaz `eth0` en el namespace del container (host networking[7]).

7 Hablaremos de los diferentes tipos de networking en Docker (host, bridge, etc) más ade-

    - Unir todos los *vethXX* creando una red *bridged* entre contenedores (*bridge* networking)
    - Para utilizar el mismo interfaz de red en varios contenedores (incluyendo loopback). Por ejemplo, un contenedor para crear una red VPN, y otro ejecutando un servicio para conectar a través de esa red. Usar la opción: `--net=container:<some_container>`

- **Ipc:** Sirve para aislar la comunicación entre procesos: semáforos, colas de mensajes, memoria compartida.
- **mnt (Mount):** Los namespaces «mount» controlan los puntos de montaje de sistemas de archivos.
    - En este caso, no se puede mover un *mount* entre namespaces (como ocurría con los interfaces de red)
    - Usos típicos:
        - Su propio rootfs (similar a `/rootfs`)
        - Puntos de montaje privados como /tmp (archivos temporales)
        - Enmascarar archivos `/proc` o `/sys`
        - Automontado de sistemas de archivos de red (NFS)
- **uts (Unix Timesharing System):** Los namespaces UTS permiten que un solo sistema parezca tener diferentes nombres de host (*nodename*) y de dominio (*domainname*) para diferentes procesos. La llamada al sistema *uname* devuelve el nodename y el domainname del proceso. [28]
- **User** (usuario): Los namespaces de usuario proporcionan tanto el aislamiento de privilegios como la segregación de identificación del usuario en múltiples conjuntos de procesos. [29]

  Usos típicos:
    - Mapear distinto UID & GID dentro del namespace. Por ejemplo

lante en el capitulo .Networking de la página 127, donde estudiaremos estas redes y otros conceptos nombrados en este punto (namespace tipo *net*).

root (UID 0) dentro del namespace, pero sin privilegios fuera.

Es una mejora en seguridad, no obstante usar UID 0 dentro del contenedor sigue siendo peligroso.

- Mapear permisos de **bind-mounts** (utilizados para cambiar la estructura del árbol de directorios). Por ejemplo, existe un usuario estándar fuera del contenedor con UID 1000, y un usuario *jenkins* dentro del contenedor con UID 1000. Podemos crear un usuario *jenkins* fuera con UID 9000, ajustando el propietario de una carpeta que se monte dentro del contenedor (con UID 9000 fuera y UID 1000 dentro de este) de forma que el usuario *jenkins* dentro del contenedor sea el propietario a pesar de tener un UID distinto.

El **UID** es el id del usuario. Los usuarios pueden combinarse para formar grupos, refiriéndose **GID** al id del grupo.

bind mount , o la opción bind del comando mount, permite que un fichero o directorio en la máquina host se monte en un contenedor.[30]

Para crear un nuevo *namespace*:

A) OPCIÓN 1: usar la llamada del sistema *clone* con un conjunto de *flags* (opciones de comando):

`CLONE_NEWPID` `CLONE_NEWUTS`

`CLONE_NEWNET` `CLONE_NEWUSER`

`CLONE_NEWIPC` `CLONE_NEWCGROUP`

`CLONE_NEWNS (Mount points)`

Véase una explicación completa sobre el uso de *clone* en el siguiente enlace:

http://man7.org/linux/man-pages/man2/clone.2.html

B) OPCIÓN 2: Usar la llamada de los sistemas

`fork` + `unshare`

Consultar Fork: http://man7.org/linux/man-pages/man2/fork.2.html

Consultar Unshare: http://man7.org/linux/man-pages/man2/unshare.2.html

Se materializarán en los siguientes pseudo archivos: `/proc/<pid>/ns`

```
root@ender:/proc/22992/ns# ls -lh
total 0
lrwxrwxrwx 1 root root 0 ene 12 21:05 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 ene 12 21:05 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 ene 12 21:05 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 ene 12 21:05 net -> net:[4026531992]
lrwxrwxrwx 1 root root 0 ene 12 21:05 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 ene 12 21:05 pid_for_children -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 ene 12 21:05 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 ene 12 21:05 uts -> uts:[4026531838]
```

Se puede cambiar de *namespace* con la llamada al sistema `setns()` o la utilidad `nsenter` (set namespace y namespace enter, respectivamente)

El namespace se elimina cuando no quedan procesos en el mismo, a no ser que se haya hecho un *`bind mount`* de la carpeta. Esto será útil para retener una referencia al namespace y reutilizarlo más tarde.

Aislamiento y KSM (Kernel Same-page merging)

Kernel Same page Merging (KSM) es una característica opcional del kernel que, una vez habilitada, permite detectar páginas que son iguales y evitar duplicaciones si múltiples procesos usan páginas de memoria que son exactamente iguales entre sí. Se utiliza sobre todo para la tecnología de virtualización basada en kernel KVM (Kernel-based Virtual Machine), de forma que múltiples máquinas virtuales con un mismo sistema operativo y mismas bibliotecas, consiguen un importante ahorro de memoria al tener muchas páginas de memoria idénticas.

El funcionamiento no es automático: además de activar esta característica en el kernel, se deben marcar las zonas de memoria a considerar candidatas para que se realice el *merging* mediante la llamada al sistema `madvise()` y poder así unirlas en una sola. En daemon de KSM (que se denomina «*ksmd*») se encarga de escanear periódicamente todas las páginas marcadas como *mergeables*, es decir, de detectar las que son iguales y unificarlas. Esto lo realiza con un mecanismo de *Copy-On-Write* [Veremos este mecanismo en profundidad en el punto de esta guía] para volver a duplicar la página en caso de que cambie desde uno de los procesos que la utilizan.

No existe un tipo de namespace para conseguir un aislamiento explícito de memoria (más allá del *ipc,* para aislar las zonas de memoria compartidas entre procesos), dado que los procesos ya tienen su propio espacio virtual de direcciones lógicas de memoria, totalmente aislado de otros procesos.

Esto quiere decir que el mismo mecanismo de KSM que se aplica por ejemplo en KVM, también se puede habilitar y utilizar con Docker o tecnologías de contenedores, ya que opera a un nivel más bajo. Sin embargo, no se suele utilizar por dos motivos:

- Primero, al requerir marcar explícitamente las páginas como *mergeables* mediante la llamada al sistema `madvise`, necesitaríamos modificar las

aplicaciones de los contenedores para marcar su memoria, o la de las bibliotecas compartidas. En KVM esto es mucho más sencillo, ya que al tratarse una virtualización a nivel hardware, la gestión de memoria es realizada por los propios procesos de KVM (el kernel no ve los procesos internos de la VM (máquina virtual), sino únicamente el proceso de emulación KVM). Por tanto, KVM puede ejecutar `madvise` sobre todas las áreas de memoria de la máquina virtual. En contenedores, podríamos usar alguna alternativa, como acceder a la biblioteca [`https://vleu.net/ksm_preload/`] Esta se carga mediante la variable de entorno `LD_PRELOAD`, reemplazando a la biblioteca C estándar y haciendo que todo bloque de memoria reservado por el proceso se marque como *mergeable*.

- La segunda cuestión está relacionada con el rendimiento. El coste de escanear periódicamente las páginas de memoria y encontrar duplicados no es despreciable (aunque los parámetros de *ksmd* se pueden ajustar). Tiene sentido, entonces, utilizar KSM solo en **escenarios en los que existe un uso muy alto de memoria y con un gran porcentaje de páginas compartidas,** ya que el ahorro será importante y merecerá la pena respecto a la pérdida de rendimiento. Sin embargo, esta sobrecarga puede no compensar en un escenario con procesos ligeros y mucha variabilidad, como suele ser más habitual en contenedores.

Control Groups (*Cgroups*)

Otra tecnología del kernel de Linux es ***Control Groups (cgroups)*** que permite establecer limitaciones en el uso de un conjunto de recursos para un proceso, compartiendo los recursos entre contenedores y estableciendo distintos tipos de límites. También permite hacer *accounting* (contabilizar el uso de recursos), `/dev` (cierto control de acceso a los ficheros de dispositivos), *congelar* grupos de procesos y otras cosas.

Los *cgroups* se activan globalmente (es decir, están habilitados para todos los procesos, o para ninguno) y son jerárquicos, con un nodo raíz por cada tipo de recurso. Véase el siguiente ejemplo de cgroup, en el que cada número representa un ID de proceso (o PID):

```
cpu                         memory
├── batch                   ├── 109
│   ├── hadoop              ├── 88
│   │   ├── 88              ├── 25
│   │   └── 109             ├── 26
└── realtime                └── databases
    ├── nginx                   ├── 1008
    │   ├── 25                  └── 524
    │   └── 26
    ├── postgres
    │   ├── 524
    └── redis
        └── 1008
```

Los cgroups se han de usar teniendo en cuenta los siguientes puntos:

- En cada recurso (cpu, memory, …) se pueden crear diferentes nodos (o subgrupos). En el ejemplo, el recurso “cpu” tiene los subgrupos “batch” y “realtime”. A su vez éstos se dividen en otros subgrupos.
- Cada proceso (PID) sólo puede estar en un nodo para cada recurso. Por ejemplo el PID 1008 está en el subgrupo “redis” del recurso “cpu”, pero no podría estar también e nel subgrupo “batch” o “hadoop”.
- Cada PID puede estar en distintos subgrupos de distintos recursos. Según el ejemplo, el PID 88 pertenece al subgrupo `batch -> hadoop` de `cpu`, y al

nodo raíz de memoria.

- El PID 1 pertenece siempre al raíz de cada jerarquía
- Nuevos procesos empiezan en el mismo nodo que su padre.

Los cgroups se pueden ver en un pseudo file system montado típicamente en `/sys/fs/cgroup:`

```
$ mount
…

cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib/systemd/systemd-
cgroups-agent,name=systemd)

cgroup on /sys/fs/cgroup/memory type cgroup
(rw,nosuid,nodev,noexec,relatime,memory)

cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup
(rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)

cgroup on /sys/fs/cgroup/devices type cgroup
(rw,nosuid,nodev,noexec,relatime,devices)

cgroup on /sys/fs/cgroup/hugetlb type cgroup
(rw,nosuid,nodev,noexec,relatime,hugetlb,release_agent=/run/cgmanager/
agents/cgm-release-agent.hugetlb)

cgroup on /sys/fs/cgroup/rdma type cgroup
(rw,nosuid,nodev,noexec,relatime,rdma,release_agent=/run/cgmanager/agents/
cgm-release-agent.rdma)

cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)

cgroup on /sys/fs/cgroup/pids type cgroup
(rw,nosuid,nodev,noexec,relatime,pids,release_agent=/run/cgmanager/agents/
cgm-release-agent.pids)

cgroup on /sys/fs/cgroup/freezer type cgroup
(rw,nosuid,nodev,noexec,relatime,freezer)


cgroup on /sys/fs/cgroup/perf_event type cgroup
(rw,nosuid,nodev,noexec,relatime,perf_event,release_agent=/run/cgmanager/
agents/cgm-release-agent.perf_event)

cgroup on /sys/fs/cgroup/cpuset type cgroup
(rw,nosuid,nodev,noexec,relatime,cpuset,clone_children)

cgroup on /sys/fs/cgroup/blkio type cgroup
(rw,nosuid,nodev,noexec,relatime,blkio)
```

Se pueden crear groups utilizando el comando `mkdir`:

```
$ mkdir /sys/fs/cgroup/memory/somegroup/subcgroup
```

Y asignar un PID a un group:

```
$ echo $PID > /sys/fs/cgroup/.../tasks
```

En cgroup.procs aparecen los PIDs de los procesos, y en `tasks` los hilos.

También existen **herramientas para gestionar**:

- **Systemd**
  - Permite especificar propiedades de cgroup en la configuración *unit* del servicio:

    ```
    [Service]
    ControlGroupAttribute=memory.swappiness 70
    ```

  - `systemctl set-property <group> CPUShares=512`
- **Cgmanager**: un daemon que interactúa con los grupos, gestionable mediante el comando `cgm`, a través de la línea de comandos
- **Libcgroup** (cgroup-tools en sistemas operativos Debian/Ubuntu)
  - `lscgroup`
  - `cgcreate`
  - `cgexec`
  - `cgset`
  - `cgdelete`
  - …

Memory

El recurso memoria *(**memory**)* provee de 3 funcionalidades a la tecnología cgroups:

- **Accounting**
  - Con *granularidad* de páginas de memoria (4kb, o dependiente de arquitectura). Esto es, la división lógica de las páginas en segmentos de dicho tamaño (4kb, o lo que corresponda).
  - Diferencia páginas que son leídas por el disco duro (que tienen copia en la memoria *swap*, y podrían ser expulsadas) o páginas anónimas.
  - Clasificación de páginas activas e inactivas (candidatas para ser expulsadas).
  - *Shared pages* (accedidas desde múltiples grupos), cuentan para uno de los grupos, a no ser que ese grupo desaparezca y pasan a formar parte de otro.
- **Limits**
  - *Hard limits:* el proceso es matado por el OOM (Out-of-Memory) killer si lo supera.
  - *Soft limits:* cuanto más supera un proceso su soft limit, más candidato es para que páginas de su memoria sean reclamadas (y mandadas a *swap*).
  - Se pueden aplicar límites a 3 tipos de memoria: física, memoria de kernel, y total.
- **Notifications**
  - Definición de umbrales, para lanzar notificaciones cuando son superados, mediante un mecanismo llamado `eventfd`.
  - También oom-notifier (Out-Of-Memory notifier*)*.

- Permiten mecanismos como lanzar una notificación y ejecutar algo que congela los procesos, para poder aumentar los límites, migrar los contenedores, etc.

El cgroup de **memory añade un pequeño *overhead* (sobrecoste, exceso de memoria)**, al tener que contabilizar los accesos y uso de las páginas de memoria.

Cpu

Las funcionalidades de las que provee el recurso **CPU** (Unidad Central de Procesamiento) a la tecnología cgroups son las siguientes:

- Accounting del tiempo de uso de CPU (user/system)
- Tiempo de uso por cada CPU
- No permite límites, pero si *weights* (pesos): El uso de pesos permite dar más prioridad a unos procesos respecto a otros.

  Por ejemplo, asignamos a un proceso un peso de 1, y a otro proceso un peso de 2. El proceso con el doble de peso tendrá el doble de tiempo de CPU disponible. Por ejemplo, suponiendo que el 99% de CPU estuviera disponible para estos dos procesos, el proceso con peso 1 dispondría de un 33% y el de peso 2 de un 66%. En caso de que el proceso con peso 2 no esté haciendo uso de la CPU, el proceso con peso 1 hará uso de la CPU completa.

Throttling o limitación de uso de CPU

¿Por qué no hacer *throttling* del recurso? Muchas CPUs modernas reducen su velocidad (es decir, hacen *throttling* automáticamente) si no se está usando la CPU al completo. El resultado es que si limitamos la ejecución de un proceso, por ejemplo a un 10%, la CPU reduciría su velocidad (ya que se ha reducido su uso), disminuyendo, así, su rendimiento. Por ello, deberemos aumentar el porcentaje de ejecución del proceso si queremos lograr el mismo rendimiento, y

hacer así que la CPU vuelva a ejecutarse más rápido.

> *Throttling*, o escalado dinámico de la frecuencia, es una técnica aplicada en arquitectura computacional, mediante la cual la frecuencia de un microprocesador CPU puede ser ajustada «sobre la marcha», en función de sus necesidades reales en cada momento. Esto permite conservar energía y reducir la cantidad de calor generado por el chip. [31]

Es por esto que no se debería hacer *throttling* de un recurso que, además, no tiene sentido limitar, ya que no causa perjuicio si se está usando al 100%. Por tanto, **es mucho más razonable trabajar con pesos o prioridades que hacer throttling de este recurso.**

¿Entonces, Docker no soporta limitar el uso de la CPU? ¿Cómo es posible que en la plataforma Kubernetes sí se pueden definir estos límites?

La respuesta es que Docker **sí que soporta** throttling, pero no utiliza *cgroups*, sino el algoritmo planificador CFS (Completely Fair Schedule) [32] del kernel de Linux , que sí permite definir, mediante un par de parámetros de Docker, límites de ejecución para los procesos. Para ampliar la información sobre ello, puede visitarse el siguiente enlace :

https://docs.docker.com/config/containers/resource_constraints/#configure-the-default-cfs-scheduler

Por tanto, los siguientes parámetros configuran los valores de **`CFS`** para el proceso, y crean límites reales de ejecución:

- `--cpu-period=<value>` - Define el periodo del *scheduler.* Por defecto 100µs (microsegundos).
- `--cpu-quota=<value>` - Define el máximo por periodo de uso de la CPU.

Sin embargo, el parámetro `--cpu-shares` establece el peso del proceso, para repartir el uso de CPU mediante prioridades.

Cpuset

El subsistema ***Cpuset*** permite controlar la asignación de CPUs, controlando la ubicación del procesador y la ubicación de la memoria de los procesos [33]. En concreto, ofrece las siguiente funcionalidades:

- Fijar un determinado set de CPUs.
- Reservar CPUs para aplicaciones específicas.
- Sistemas NUMA (Non Uniform Memory Access), en los que cada CPU tiene bancos de memoria dedicados.

Block I/O [Blkio]

El subsistema Block I/O (Input/Output) [`Blkio`] implementa el controlador de operaciones de bloque input-output, estableciendo diferentes políticas de control y limitación y proveyendo así a cgroups de las siguientes funcionalidades [34]:

- Accounting o medición de uso
    - Por cada dispositivo de bloque
    - En operaciones de lectura vs escritura (*reads* vs *writes*)
    - En operaciones con sincronía vs asincronía (*sync* vs *async*)
- Límites / throttling
    - Por cada dispositivo de bloque
    - En operaciones de lectura vs escritura (reads vs writes)
    - Ops (número de operaciones) vs bytes leídos o escritos.
    - Como la mayoría de las escrituras son asíncronas, puede parecer que los límites no funcionan porque las escrituras se hacen primero a la caché (ya que escribe a memoria, y es el *flushing* lo que tiene el throttling).

Network class [Net_cls] y Network priority [Net_prio]

Los subsistemas Network class [**Net_cls**] y Network priority [**Net_prio**] permiten establecer un tipo de *clase* [**Net_cls**] o de prioridad, [**Net_prio**] al tráfico generado en este contenedor. Únicamente al tráfico saliente (*egress*), ya que es complicado ejercer control sobre tráfico entrante, pues este ya ha llegado al contenedor, por lo que no tiene sentido descartarlo.

- `Net_cls` asigna o etiqueta tráfico a una clase, pero luego se debe usar la utilidad `tc` (traffic control) o reglas de iptables (las veremos más adelante, apartado IPTables y modo bridge pag. 128) o no habrá ningún efecto sobre el tráfico, que por defecto fluye de forma normal.
- `Net_prio` asigna una prioridad de tráfico, y de nuevo se requiere usar `tc` o similar.

Devices

El recurso **devices** **[/dev]** permite establecer qué dispositivos (device nodes, **/dev**) puede leer, escribir o crear (usando en este caso el comando `mknod`) el cgroup. Resulta muy útil para prevenir la lectura/escritura directa de discos (`/dev`/sdX, etc)

- Normalmente el recurso **devices,** permite el acceso a lo siguiente (denegando el acceso a todo lo demás):
  - `/dev/tty`: Terminal del proceso actual.
  - `/dev/zero`
  - `/dev/random`: Para no agotar la nula entropía existente dentro de un contenedor, el acceso a este dispositivo permite el acceso a la generación de datos aleatorios del *host*.
  - `/dev/null`

- Otros:
    - `/dev/net/tun`: Para crear VPNs o similar en un container (dispositivos tunel))
    - `/dev/fuse`: Sistemas de archivos en espacio de usuario
    - `/dev/kvm`: Para ejecutar máquinas virtuales
    - `/dev/dri` & `/dev/video:` Para acceso a GPU (unidades de procesamiento gráfico), minado de bitcoins, etc.

Freezer

Por último, el recurso **freezer** permite congelar un cgroup completo sin tener que usar las señales `SIGSTOP` / `SIGCONT` (que interfieren en el proceso), pudiendo detectar e interceptar la señal, e interfiriendo con ella si está depurando con la llamada al sistema `ptrace`. También permite cosas como *job scheduling*, deteniendo y ejecutando periódicamente, o incluso la migración de contenedores, parando los procesos, haciendo un checkpoint, y rearrancandolos en otra ubicación.

> Ptrace (process trace, o traza del proceso): llamada al sistema a través de la cual un proceso puede observar y controlar la ejecución de otro proceso y examinar y cambiar la memoria y los registros del mismo. [35]

Para ampliar la información acerca de esta manera de migrar contenedores, consúltese el siguiente link:

https://www.criu.org/

Almacenamiento

En Docker, las imágenes se construyen a base de capas, en las que cada capa contiene únicamente las diferencias que hemos añadido respecto a la capa padre. Para montar en una carpeta la combinación de las distintas capas, Docker utiliza mecanismos de «*union filesystems*» o bien subvolúmenes, *snapshots* (copia instantánea de volumen) u otras características avanzadas de DeviceMapper o sistemas de archivos tipo BTRFS, XFS, etc. (los veremos más adelante).

Al crear un contenedor, Docker añade una capa adicional (la capa de contenedor), que es la única sobre la que es posible escribir. De esta forma el contenedor puede **modificar** aparentemente la imagen base, como si tuviera una copia real, aunque en realidad está modificando solo esta última capa. Así, como vemos en el siguiente esquema de la Figura 15 podemos crear múltiples contenedores sobre una misma imagen (ubuntu:15.04 Image), reutilizando todas las capas excepto la capa Read/Write de contenedor (thin R/W layer).

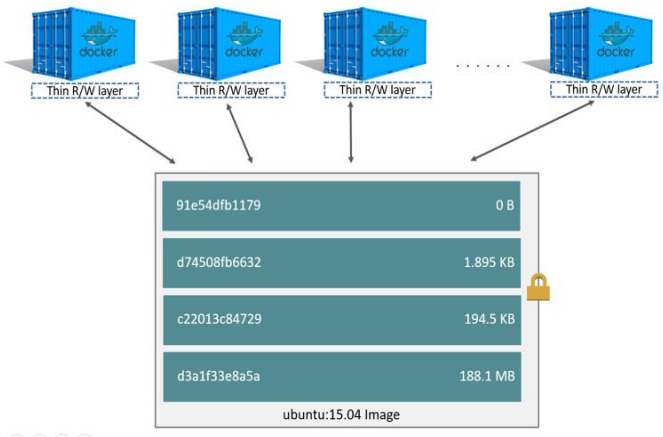

Figura 15: Múltiples contenedores de la imagen « ubuntu 15.04 » en Docker

Al destruir un contenedor, la capa con las modificaciones se destruye, a no ser que la convirtamos en una nueva imagen con el comando **`docker commit`**.

Además, el comando `docker commit` nos permitirá, a la vez que generamos una nueva imagen, cambiar la configuración del entorno del contenedor (como variables de entorno definidas con `ENV`) en el Dockerfile, el `CMD`, el `ENTRYPOINT`, `LABELS`, `USER`, los puertos expuestos mediante `EXPOSE`, etc.

Véase el siguiente enlace para más información:

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

Copy-on-write

El mecanismo **copy-on-write (CoW)** permite crear contenedores instantáneamente, en lugar de tener que hacer una copia completa del sistema de archivos base de la imagen. El sistema de almacenamiento lleva el control de **lo que ha cambiado** respecto al sistema de archivos base. Esto puede ser implementado con:

- AUFS (overlay filesystem)
- El *framework* mapeador de dispositivos Device Mapper, que implementa el método de optimización de eficiencia *thin provisioning* (capas de almacenamiento a nivel de dispositivo de bloque)
- El sistema de archivos BTRFS
- El sistema de archivos y administrador de volúmenes ZFS

Veremos todo esto en detalle en los siguientes apartados.

Así, a través de este mecanismo «copy-on-write», podemos tener multitud de contenedores sin ocupar ningún espacio adicional, mientras no escribamos cambios dentro del contenedor.

Permite también hacer cosas como usar el comando `docker commit` para generar una nueva imagen base con los cambios realizados en un determinado contenedor.

El funcionamiento es similar al Copy-On-Write para la memoria, usado por las llamadas al sistema `fork`, `mmap`, etc. donde el dispositivo MMU (*Memory Management Unit*) traduce direcciones lógicas dentro de un proceso con páginas de memoria (normalmente 4kb) en la memoria física (o *page fault*, fallo de página), puede estar en la memoria *swap*, segmentation *fault* (violación de acceso), etc. Cuando se produce una escritura en una página copiada (en el *fork* o mapeada en el *mmap*), se ejecuta la copia real, y se vuelve a lanzar la escritura, esta vez sobre la página copiada.

El Copy-On-Write para discos se utilizaba principalmente para hacer copias instantáneas de volumen (snapshots), o backups consistentes de una BD (base de datos), asegurándose de que nada cambia entre el inicio y el final de la copia. Luego se comenzó a usar para el método «thin provisioning» en VMs (máquinas virtuales), consiguiendo que una imagen de disco se pueda reutilizar sin tener que realizar copias completas, snapshots, etc.

Este mecanismo también es usado por:

1. LVM (gestor de volúmenes lógicos)
2. ZFS
3. BTRFS
4. AUFS, Overlayfs, UnionMount
5. Discos virtuales (VHDs) en VMs

> *UnionMounts*, o montajes de unión es una forma de combinar múltiples directorios en uno que parece contener sus contenidos combinados. [36]
>
> Un *disco duro virtual, o VHD* por sus siglas en inglés, es una emulación virtual de un disco duro de ordenador que, gracias a la conexión a Internet, permite el acceso al mismo desde cualquier lugar.[37]

Storage Driver

El driver de almacenamiento (*storage driver*) determina la implementación usada por Docker de entre las que hemos visto en el apartado anterior para el mecanismo de copy-on-write.

La selección del *storage driver* viene normalmente limitada por la disponibilidad de ciertas características, como el soporte de sistemas de archivos, de la distribución y el kernel sobre el que se ejecute Docker.

Se puede ver el driver utilizado con el siguiente comando [en este caso `overlay2`, profundizaremos sobre este más adelante, en el apartado OverlayFS (overlay2) pag. 91)] :

```
$ docker info

Containers: 0
Images: 0
Storage Driver: overlay2
 Backing Filesystem: xfs
...
```

Y se puede cambiar el driver actuando sobre los parámetros de inicio de dockerd. No obstante, conviene revisar la documentación específica del driver, ya que algunos requieren configuración adicional.

La documentación oficial de Docker contiene también otras recomendaciones y consideraciones a tener en cuenta, véanse todas en el siguiente link:

https://docs.docker.com/storage/storagedriver/select-storage-driver/

Drivers disponibles

A continuación, veremos más en profundidad varios de los **drivers** mencionados que están **disponibles para la implementación del mecanismo copy-on-write**, siendo todos ellos tipos de sistemas de archivos llamados «*union filesystems*».

Esto significa que múltiples capas de una imagen, almacenadas en distintos

directorios dentro del mismo host Linux, se presentan en un punto de montaje como un único directorio, combinando el contenido de todas las capas.

- AUFS
- OverlayFS (overlay2)
- Device Mapper
- BTRFS y ZFS
- VFS

1 AUFS

dotCloud (que, recordemos, es la compañía que desarrolló Docker como "platform-as-a-service") empezó a usar **AUFS** (versión alternativa de sistemas de archivos de unión para Linux) antes de desarrollar Docker, y aunque su implementación no estaba exenta de problemas y complejidades, funcionaba bien y se usaba en otros proyectos como los LiveCDs de Debian y Ubuntu, donde el sistema de archivos se monta con Copy-On-Write AUFS sobre el sistema de archivos de sólo lectura del CD/DVD.

El sistema de archivos AUFS sigue siendo el driver preferido para versiones de Docker inferiores a la versión 18.06, cuando se ejecutan en Ubuntu 14.04, cuyo kernel no tiene soporte para *overlay2* (driver que veremos en el siguiente apartado).

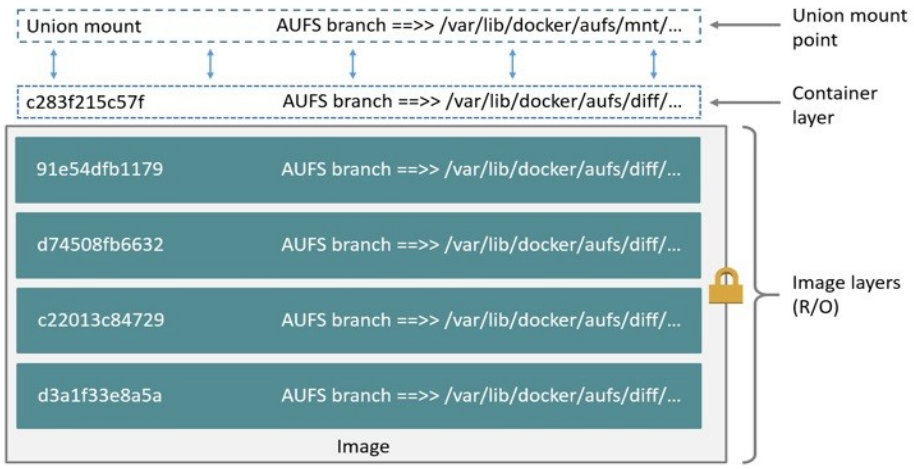

Figura 16: Uso de AUFS

En AUFS, se denomina *branch* (AUFS branch) al equivalente de capa de Docker. Cada capa de una imagen *(image layer),* y la capa read/write del contenedor, se representa como subdirectorios en `/var/lib/docker/aufs`, proveyendo el *union mount* la vista unificada de todas las capas, tal y como se muestra en la Figura 16.

Es importante notar que los nombres de los directorios no se corresponden con los IDs de las capas en Docker, siendo:

- `/var/lib/docker/aufs/mnt/$CONTAINER_ID/`
  para la capa del contenedor
- `/var/lib/docker/aufs/diff/…`
  para las capas de las imágenes

AUFS usa Coppy-on-Write (CoW) para hacer más eficiente el almacenamiento y el rendimiento, de la siguiente manera:

- Cuando se lee un fichero, éste se busca en la capa superior (*container layer*), y en caso de no encontrarse se busca en las capas inferiores

  [OJO: El rendimiento de AUFS sufre mucho cuando se intenta buscar ficheros que no existen en contenedores con muchas capas].

- Cuando se escribe un fichero que no existe en la capa superior, se hace una operación *copy-up* del fichero entero (no se trabaja a nivel de bloque), y los cambios se hacen sobre esa copia.

- El borrado se realiza escribiendo un fichero que indica «*borrado»* en la capa superior (también para directorios).

Para más información a cerca de AUFS consultar el enlace:

https://docs.docker.com/storage/storagedriver/aufs-driver/

2 OverlayFS (overlay2)

OverlayFS [overlay 2] Es un union filesystem similar a AUFS, pero más rápido y con una implementación más sencilla. Sin embargo, tiene requisitos más

exigentes, como un kernel 4.0 o superior, y algunas configuraciones específicas en algunos sistemas de archivos como XFS.

> XFS es un sistema de archivos de alta escalabilidad de 64 bits. Se basa totalmente en la extensión, por lo que soporta archivos y sistemas de archivos muy grandes. El número de archivos que pueden contener un sistema XFS está limitado únicamente por el espacio disponible en el sistema de archivos. [38]

OverlayFS (luego renombrado simplemente a overlay) sólo puede tener dos *branches* (*upper* y *lower*). Pero la capa *lower* puede a su vez ser un *overlay*.

Permite unir hasta 128 «`lowerdir`» (lower directories) con un «`upperdir`» (upper directories) en el mount point llamado «`merged`» (directorio que muestra una vista unificada de los dos anteriores).

En el siguiente diagrama de la Figura 17 se muestra cómo las construcciones de Docker se asignan a las construcciones de OverlayFS. La capa de la imagen (image layer) es el «`lowerdir`», la capa del contenedor (container layer) el «`upperdir`», y el directorio «`merged`» es el punto de montaje de los contenedores (container mount). [39].

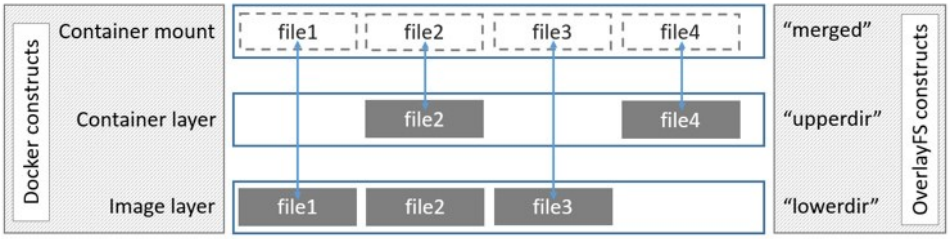

Figura 17: Construcciones por capas en Docker frente a OverlayFS

Las capas se almacenan en

`/var/lib/docker/overlay2/$BRANCH_ID`.

> Hay una carpeta “L minúscula" especial con enlaces simbólicos para evitar problemas con la máxima longitud de parámetros al comando mount.

La capa más baja de todas contiene un subdirectorio `diff/` con los contenidos, y otro `link/` con el nombre abreviado de la capa. Las siguientes capas contienen un directorio `diff/` similar, con los contenidos de la capa, otro `merged/` que

unifica sus contenidos y los de la capa padre, y otro `work/` usado internamente por overlay2, así como un fichero *lower-id* indicando el padre.

> La versión previa (overlay, sin el "2") sólo soportaba una *lower layer*, por lo que no soportaba múltiples capas, y se usaba un mecanismo de *hard-linking* (asocian dos o más ficheros compartiendo el mismo *i-nodo*) para crear el contenido de `merged/`. El problema que esto generaba era un alto consumo de *i-nodos* en el sistema de archivos.

El funcionamiento de las lecturas y escrituras es muy similar a AUFS, con la diferencia de que OverlayFS sólo funciona con dos capas (la del contenedor, y la merged/ de la capa padre), por lo que la búsqueda de archivos es más eficiente.

Para ampliar información acerca de OverlayFS:

https://docs.docker.com/storage/storagedriver/overlayfs-driver/

3 Device Mapper

La compañía Redhat se interesó en Docker y comenzó a utilizarlo contribuyendo con el driver «**Device Mapper**», para usar LVM (gestor de volúmenes lógicos). **DeviceMapper** era el driver por defecto en Redhat y CentOS, ya que el kernel no tenía soporte para overlay2, el driver recomendado. Sin embargo, este driver requiere acceso directo a los dispositivos LVM para un rendimiento óptimo y para su uso en producción. Se puede usar, entonces, *loopback-lvm* (montar dispositivos LVM sobre la interfaz de red virtual loopback), sin configuración adicional, pero es lento y no recomendado.

Realmente en Docker se utiliza LVM por su capacidad de *Thin Provisioning* (thinp). La configuración de LVM para producción es algo más compleja. Se puede pre-crear el dispositivo LVM, o bien usar el modo `direct-lvm` sobre un dispositivo físico y dejar que Docker lo gestione.

En la Tabla 2 se muestran las carpetas que contienen información importante cuando se usa DeviceMapper:

Tabla 2: Contenido de carpetas Device Mapper

Carpeta	Contenidos
`/var/lib/docker/devicemapper/metadata/`	contiene metadatos sobre la configuración de devicemapper y las distintas capas
`/var/lib/docker/devicemapper/mnt/`	contiene puntos de montaje para cada imagen y para las capas de contenedor.
`/var/lib/docker/devicemapper/mnt/$CONTAINER_ID/`	Punto de montaje del contenedor.

Cada capa es un *snapshot* de la capa padre. La capa más baja de cada imagen es un snapshot del dispositivo base (*base device*) que existe en el grupo (*pool*). Cuando se ejecuta un contenedor, es un snapshot de la imagen en la que se basa el contenedor. El siguiente ejemplo en forma de diagrama de la Error: no se encontró el origen de la referencia muestra un host Docker con dos contenedores en ejecución. El primero es un contenedor del sistema operativo Ubuntu y el segundo de Busybox:

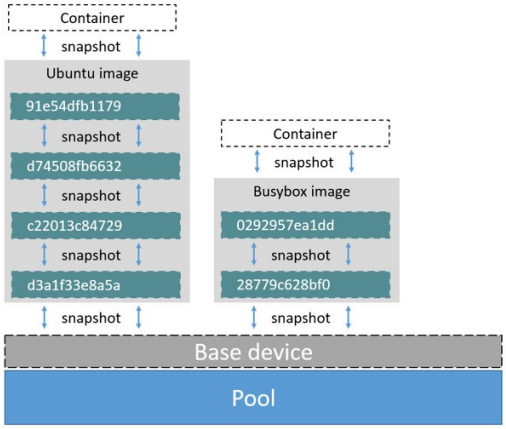

Figura 18: Host ejecutando contenedores Ubuntu y Busybox

El principal cambio que ofrece **Device Mapper** respecto a *union filesystem* es que las lecturas y escrituras se realizan a nivel de **bloque**, no de archivo. En ciertas circunstancias esto puede ser beneficioso, pero en otras no. Por ejemplo, el tamaño de bloque mínimo es de 64KB, por lo que una escritura siempre consumirá como mínimo ese espacio, aunque se escriba un fichero de menor tamaño. Sin embargo, puede ser útil si se modifica una pequeña parte de un archivo grande, ya que no requiere un *copy-up* del archivo completo.

Para ampliar información a cerca de Device Mapper:

https://docs.docker.com/storage/storagedriver/device-mapper-driver/

4 BTRFS y ZFS

BTRFS (B-tree File System) **y ZFS** (Zettabyte File System) son los drivers utilizado si el sistema de archivos sobre el que se utiliza Docker es alguno de estos, ya que soportan creación de subvolúmenes con Copy-On-Write «out-of-the-box», y opciones avanzadas como snapshots.

Driver BTRFS

`BTRFS` usa una característica del sistema de archivos llamada subvolúmenes, almacenando un directorio por cada imagen o capa de contenedor dentro de la carpeta localizada en:

```
/var/lib/docker/btrfs/subvolumes/
```

Los subvolúmenes son un mecanismo de copy-on-write, que pueden ser anidados, y sobre los que se pueden generar snapshots. Sólo la capa base se almacena como un auténtico subvolumen. Todas las demás capas se almacenan como snapshots. La localización para cada capa o contenedor es:

```
/var/lib/docker/btrfs/subvolumes/$CONTAINER_OR_IMAGE_ID/
```

En la Figura 19 se observa de qué manera las construcciones de Docker se asignan, en este caso, a las construcciones de BTRFS.

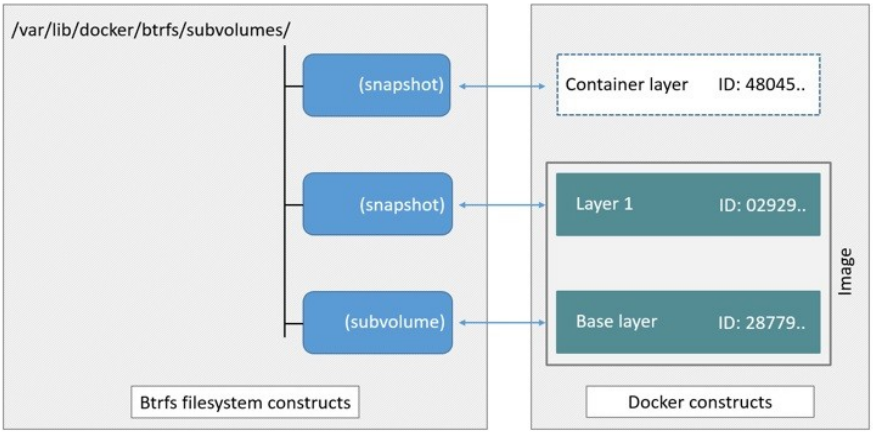

Figura 19: Asignación de construcciones de Docker a construcciones de BTRFS

El proceso de creación de imágenes y contenedores en el host Docker que ejecuta el controlador es el siguiente:

1. La capa de base de la imagen se almacena en /var/lib/docker/btrfs/subvolumes
2. Las capas posteriores se almacenan como snapshots con los cambios introducidos en cada capa. Estas diferencias se almacenan en el nivel de bloque.
3. La capa de escritura del contenedor es un snapshot de la capa final de la imagen, donde el contenedor en ejecución introduce las diferencias, que se almacenan en el nivel de bloque.

La gestión se hace también a nivel de bloque, no de archivo, por lo que en ese sentido es muy similar al funcionamiento de DeviceMapper. En el disco, los snapshots se ven y actúan como subvolúmenes, pero en realidad son mucho más pequeñas y más eficientes en cuanto al espacio. Copy-On-Write se utiliza para maximizar la eficiencia del almacenamiento y minimizar el tamaño de la capa, y las escrituras en la capa gravable del contenedor se administran a nivel de bloque. La siguiente imagen muestra la compartición de datos entre un subvolumen BTRFS y sus snapshot. [40]

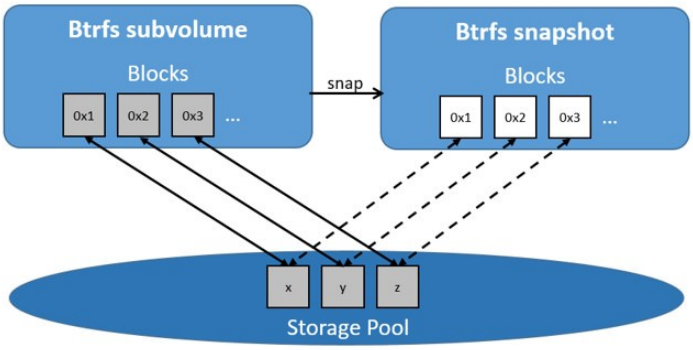

Figura 20: Subvolumen BTRFS y su snapshot compartiendo datos

Uno de los problemas de BTRFS es que gestiona *chunks (*parte de un grupo de bloques), por lo que podríamos quedarnos sin espacio aunque la respuesta al comando `df` muestre que existe espacio disponible (al no estar los chunks completamente ocupados).[41]

```
# btrfs filesys balance start -dusage=1 /var/lib/docker
```

Más información sobre BTRFS:

https://docs.docker.com/storage/storagedriver/btrfs-driver/

Driver ZFS:

ZFS también funciona de forma similar, mediante clonados (clone) y snapshots. La capa base de una imagen es un sistema de archivos ZFS. Cada capa hija es un clonado ZFS basado en un snapshot de la capa inferior. Un contenedor es un clonado ZFS basado en el snapshot de la capa superior de la imagen a partir de la cual se crea.

Cuando se inicia un contenedor, los siguientes pasos ocurren en el orden siguiente (Figura 21) [42]:

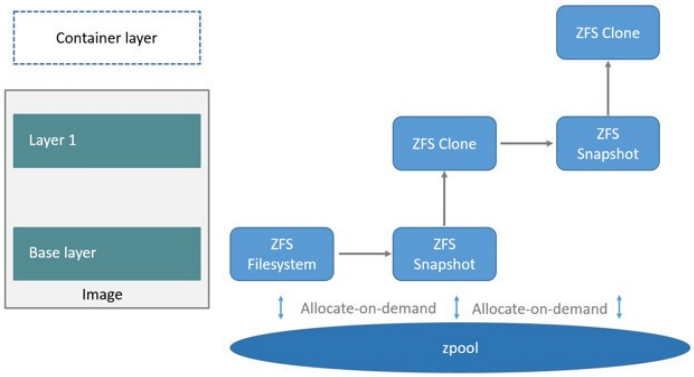

Figura 21: Funcionamiento de ZFS para un contenedor basado en una imagen de dos capas.

1. La capa base de la imagen existe en el host Docker como un sistema de archivos ZFS.
2. Las capas de imagen adicionales son clones del conjunto de datos que aloja la capa de imagen directamente debajo de ella.
3. En el diagrama, se agrega "Capa 1" (layer 1) al tomar un snapshot ZFS de la capa base (base layer) y luego se crea un clon a partir de ese snapshot. El clon es modificable y consume espacio a demanda del zpool. La instantánea es de lectura unicamente, manteniendo la capa base como un objeto inmutable.
4. Cuando se inicia el contenedor, se agrega una capa modificable sobre la imagen.
5. En el diagrama, la capa de lectura y escritura del contenedor se crea haciendo una instantánea de la capa superior de la imagen (Capa 1) y creando un clon a partir de esa instantánea.

6. A medida que el contenedor modifica el contenido de su capa modificable, se asigna espacio para los bloques que se cambian. Por defecto, estos bloques son 128k.

La gestión es a nivel de bloque, igual que en BTRFS y DeviceMapper.

Más información sobre ZFS:

https://docs.docker.com/storage/storagedriver/zfs-driver/

5 VFS

El driver de almacenamiento **VFS** sólo es recomendado para tests y pruebas, y no en entornos de producción reales, ya que su rendimiento es pobre. Se puede utilizar solo si no existe soporte para ningún tipo de sistema de archivos con copy-on-write.

Características:

- VFS no es un driver tipo «*union filesystem*» como AUFS o OverlayFS.
- Cada capa de la imagen y la capa que se puede escribir del contenedor se representan como un subdirectorio en: `/var/lib/docker/vfs/dir/`.
- El punto de montaje de unión (mount point) da una visión unificada de las capas.
- No se soporta copy-on-write, por lo que cada vez que se crea una nueva capa se hace una copia completa de la capa padre.

Para ampliar la información sobre VFS:

https://docs.docker.com/storage/storagedriver/vfs-driver/

Content Addressable Storage (CAS)

Content Addresssable Storage (CAS) es una tecnología de almacenamiento que aporta un mecanismo rápido y eficiente para almacenar y recuperar datos fijos en el disco. Al igual que el mecanismo copy-on-write visto en el punto anterior, la tecnología CAS perite hacer un mejor uso de la memoria y optimizar el almacenamiento en Docker, caracterizándose porque, cada capa en una imagen Docker y **la capa de contenedor, se identifican mediante un ID único.**

Antes de la versión 1.10 de Docker, este ID era un UUID (identificador único universal) generado aleatoriamente (Random UUID). Desde la versión 1.10 y superiores se utiliza un mecanismo llamado «*secure content hash*» que mejora la seguridad, evita colisiones, garantiza la integridad de los datos al hacer `pull/push`, `load/save`, etc., y, además, mejora la compartición de capas, permitiendo que muchas imágenes que están construidas sobre una misma base compartan sus capas sin ocupar espacio adicional, incluso si no vienen del mismo `build`.

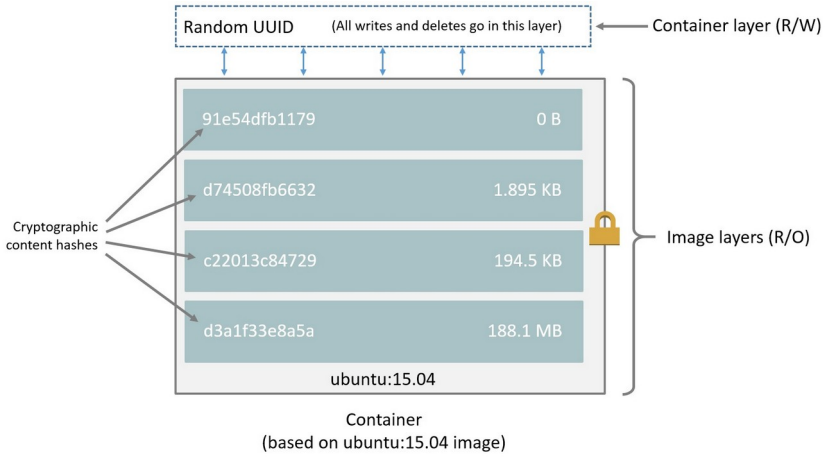

Figura 22: Contenedor basado en imagen de Ubuntu 15.04

El diagrama de la Figura 22 muestra un contenedor basado en la imagen de Ubuntu 15.04 (que comprende 4 capas de imágenes apiladas, o *image layers*).

En la capa superior de la pila subyacente, se agrega una nueva capa delgada y modificable (container layer). Todos los cambios realizados en el contenedor en ejecución (writes and deletes) se escriben en esta delgada capa de contenedor modificable. [43]

Las capas se identifican por su ***digest*** (esto es, el ID de la imagen en Docker), que tiene la siguiente forma:

```
algoritmo:hexadecimal
```

Por ejemplo:

```
sha256:fc92eec5cac70b0c324cec2933cd7db1c0eae7c9e2649e42d02e77eb6da0d15f
```

En este caso, el algoritmo utilizado es el llamado SHA256 (*Secure Hash Algorithm 256*), que genera este *digest* de 256 bits (32 bytes) a modo de *firma.*

El *digest* se calculará aplicando dicho algoritmo al contenido de las capas. Por tanto, si cambia el contenido de estas, también cambiará el *digest*, lo que permitirá a Docker verificar la integridad de la imagen después de hacer `push` o `pull` de una capa (subir o descargar, respectivamente).

Las capas **no tienen noción de pertenencia a una imagen**, sino que son independientes de esta, siendo únicamente una colección de archivos y directorios.

A continuación mostramos en la Figura 23 un ejemplo de compartición de capas entre diferentes imágenes, procedente del sitio web imagelayers.io. Esto nos puede resultar útil para armar una composición de varios contenedores.

Otro servicio muy útil de cara a analizar el contenido y las capas de las imágenes, es el sitio web microbadger.com. Podemos probar, por ejemplo, la imagen “ubuntu”, “golang” o “java” en los siguientes links:

https://microbadger.com/images/ubuntu
https://microbadger.com/images/golang
https://microbadger.com/images/java

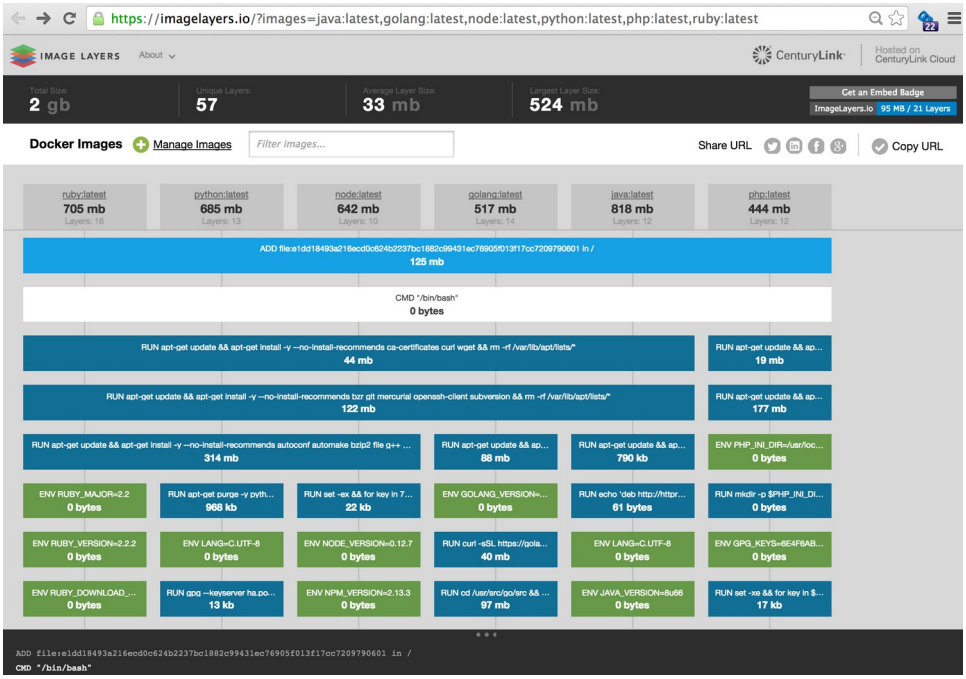

Figura 23: Ejemplo de capas compartidas entre diferentes imágenes desde imagelayers.io

Sin embargo, para guardar los comandos usados para generar cada capa, se requiere del uso de la instrucción `LABELS` en el Dockerfile, con los etiquetas correspondientes por convención (véase apartado de esta guía), según se indica en microbadger.com/labels y siguiendo el estándar indicado en el sitio[8].

8 http://label-schema.org/rc1/

Detalles de funcionamiento del Content Addressable Storage:

Una imagen Docker, al final, se compone de un fichero de configuración que contiene, entre otras cosas, una una lista ordenada de *digest* de las capas de la imagen. Este fichero de configuración, a su vez, también está indexado mediante un digest de su contenido que, de hecho, se corresponde con el *ImageId,* el identificador de la imagen. De esta manera, Docker puede componer el sistema de archivos de un contenedor con las referencias de los digests de las capas.

> Es importante entender que, en el «Content Addressable Storage», si bien las capas se identifican mediante este *secure digest*, las carpetas donde se guardan los contenidos (dependientes del driver) siguen utilizando ***CacheIDs*** generados aleatoriamente. Docker guarda una referencia entre cada capa, y también el *CacheID* correspondiente para localizar los contenidos en el disco. El motivo por el cual se realiza esta diferenciación entre los *digests* de las capas y su *CacheID*, que además es distinto en cada host (si realizamos `pull` en dos Docker daemon distintos, se generarán distintos *CacheIDs*), no está muy claro, aunque en algunos documentos se indica que es por seguridad.

Si comparamos la salida mostrada tras ejecutar el comando `docker inspect` (tal y como hicimos en el Dockerfile) y el comando `docker history`, podemos ver la correspondencia entre las distintas capas que componen la imagen y los pasos que se ejecutaron en el Dockerfile para construirla, aquí un ejemplo:

```
# docker inspect mysql:5.6
[
    {
        "Id": "sha256:742f7d5a4104969fcac8054cf9201f5656096f0a58d 10947a4a41a8e1d7d9f91",
        "RepoTags": [
            "mysql:5.6"
        ],
        "RepoDigests": [
       "mysql@sha256:9527bae58991a173ad7d41c8309887a69cb8bd178234 386acb28b51169d0b30e"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2020-01-14T23:22:00.768613912Z",
        "Container": "e688d8a03f28ea18082da4fcd0f504d17ffa5c6403 a07fb0b8 656e5cb02cbb6b",
          },

        …

        "RootFS": {
            "Type": "layers",
            "Layers": [

                "sha256:814c70fdae62bc26c603bfae861f00fb1c77fc0b1ee8d565717846f4df24ae5d",
 "sha256:25575e327c84a7dc1cf4fe937763159f5cf0388ab8847ce98343377a3e6c61b6",
                "sha256:61cb1c0dec27ba5526dddaf67a05e9fe386d72fd56fc2da3485580597ed1285f",
                "sha256:955b4c88a6e8be640471e2643d79451242c96a8efeacf9e74b93a33eeab2c9aa",
                "sha256:fef9e518b701f500c8956310954a06fc7a9960c4174361604edbb01296904e18",
                "sha256:0424189a0dcff726355204b746ae733b600fac6302b0c9e945b69f1a87bfaea3",
                "sha256:e941e4d8f5a15510486b86130ca9aef162907a2949cffa8dde9f70ac99824140",
                "sha256:6555cd8ff212cffc85b557dcf7c6ba0fccb033c5f1e8a478beb87f0cfdae8082",
                "sha256:1543269d6b47a222b56c03b5ff28a007c79228758b637302b712162707d70c31",
                "sha256:7c78f0e833ba5af88a20eb111a10a95504e35ac85dde049bef780a1e3c9c614c",
                "sha256:1d35102c0a57710a7823fbd392a896c8ef83e2c8cda582a75fafc9d106cb4d11"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```

```
# docker history mysql:5.6

IMAGE          CREATED                 READED BY
SIZE                   COMMENT
742f7d5a4104   4 days ago              /bin/sh -c #(nop)  CMD ["mysqld"]
0B

<missing>              4 days ago      bin/sh -c #(nop)  EXPOSE 3306
0B

<missing>              4 days ago      /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry…
      0B

<missing>              4 days ago      /bin/sh -c ln -s usr/local/bin/docker-entryp…    34B

<missing>              4 days ago      /bin/sh -c #(nop) COPY file:b3081195cff78c47…12.7kB
```

```
<missing>             4 days ago      /bin/sh -c #(nop)  VOLUME [/var/lib/mysql]
0B

<missing>             4 days ago      /bin/sh -c {   echo mysql-community-server m…
192MB

<missing>             4 days ago      /bin/sh -c echo "deb http://repo.mysql.com/a…   56B

<missing>             4 days ago      /bin/sh -c #(nop)  ENV MYSQL_VERSION=5.6.47-…   0B
<missing>             3 weeks ago     /bin/sh -c #(nop)  ENV MYSQL_MAJOR=5.6
0B

<missing>             3 weeks ago     /bin/sh -c set -ex;  key='A4A9406876FCBD3C45…
30.2kB

<missing>             3 weeks ago     /bin/sh -c apt-get update && apt-get install…
39.9MB

<missing>             3 weeks ago     /bin/sh -c mkdir /docker-entrypoint-initdb.d
0B
<missing>             3 weeks ago     /bin/sh -c set -x  && apt-get update && apt-…
4.44MB
<missing>             3 weeks ago     /bin/sh -c #(nop)  ENV GOSU_VERSION=1.7
0B
<missing>             3 weeks ago     /bin/sh -c apt-get update && apt-get install…
10.2MB
<missing>             3 weeks ago     /bin/sh -c groupadd -r mysql && useradd -r -…
329kB
<missing>             3 weeks ago     /bin/sh -c #(nop)  CMD ["bash"]
0B
<missing>             3 weeks ago     /bin/sh -c #(nop) ADD file:90a2c81769a336bed…
55.3MB
```

El fichero de configuración obtenido mediante `docker inspect` nos indica que hay 11 layers. Sin embargo, hay más de 11 pasos en el resultado de `docker history`. Esto se explica porque no todos los pasos generan una nueva capa, ya que algunos de ellos alteran la configuración (`CMD`, `ENV`, `ENTRYPOINT`, `EXPOSE`, etc.). Sólo hay 11 pasos que añadan archivos o modifiquen el sistema de archivos, que son los que generan las 11 capas.

Si nos fijamos en la salida del comando `docker history` nos puede parecer poco afortunada. El mensaje de «*missing*» parece significar que se trata de un error: pero no lo es. Lo que ocurre es que no existe ninguna imagen (ID de imagen) que se corresponda con esa capa. El ID de imagen `742f7d5a4104` correspondiente a mysql:5.6 se corresponde con **todo el conjunto de capas** (no

solo con la última capa) que, unidas, componen el sistema de archivos raíz de esa imagen.

El comportamiento cambia un poco cuando **construimos** una imagen localmente. Por ejemplo:

```
$ docker history airadier/localimage
IMAGE               CREATED             CREATED BY
SIZE                COMMENT

bb8ea8f03936        8 seconds ago       /bin/sh -c #(nop)  CMD ["/bin/sh -c 'echo Ho…
0B
a6a9352d7817        8 seconds ago       /bin/sh -c apk add jq
1.14MB
2535a3cd6067        9 seconds ago       /bin/sh -c apk add curl
2.92MB
e7d92cdc71fe        40 hours ago        /bin/sh -c #(nop)  CMD ["/bin/sh"]
0B
<missing>           40 hours ago        /bin/sh -c #(nop) ADD file:e69d441d729412d24…
5.59MB

$ docker images -a
REPOSITORY          TAG                 IMAGE ID        CREATED
SIZE
<none>                      <none>              a6a9352d7817    About a minute ago
9.65MB
airadier/localimage   latest              bb8ea8f03936    About a minute ago
9.65MB
<none>              <none>              2535a3cd6067    About a minute ago
8.52MB
alpine              latest              e7d92cdc71fe    40 hours ago
5.59MB
mysql               5.6                 42f7d5a4104     4 days ago
302MB
...
```

En este caso, cada paso de la imagen que hemos construido sí que ha generado imágenes intermedias (2535a3cd6067 y a6a9352d7817), que de hecho aparecen en la lista de imágenes si usamos el **flag** `-a`: -a, - -all Muestra todas las imágenes (por defecto muestra solo las imágenes intermedias) [44]

> Los *flags* proporcionan opciones a los comandos. Un flag con dos guiones delante es el nombre completo del flag. Un flag con un solo guión es un acceso directo al nombre completo del mismo. Por ejemplo, -a es la abreviatura de la bandera --all

Estas imágenes intermedias tienen su correspondiente archivo de configuración. No tienen un nombre asignado (el repositorio y tag aparecen como `<none>:<none>`), pero sí que contendrán la información del ID de la imagen padre.

Estas capas intermedias, con la referencia a sus padres, son importantes para facilitar el uso de la caché de Docker, una característica importante que permite reutilizar el contenido de capas pre-existentes al ejecutar comandos idénticos durante la construcción de nuevas capas, en lugar de generar capas nuevas desde cero.

Cuando se hace `push` de la imagen a un registro (subir dicha imagen), sólo se envía la última imagen, con las capas que la constituyen. Por lo tanto, un `pull` (descargar) en otro host distinto no contendrá las imágenes intermedias.

Ejercicio 1: qué ocurre cuando hacemos docker pull

> Sugerencia para la resolución del ejercicio: ir anotando aparte los diferentes IDs para facilitar su seguimiento durante la resolución del ejercicio (véase la Tabla 3: Tabla de ayuda para la resolución de ejercicios en la página 124)

Partiremos de lo siguiente:

```
# pwd
/var/lib/docker/image/overlay2
```

> El comando `pwd` (print working directory) devuelve la ruta en la que estamos situados dentro de la estructura de directorios, lo cual nos facilita movernos por dicha estructura sin perder nunca el control de nuestra situación en la misma. [45]

En el directorio `/var/lib/docker/image` se almacenan las imágenes de Docker.

Analicemos qué podemos encontrar en ese directorio:

```
# cat repositories.json
{"Repositories":{}}

# ls imagedb/content/sha256/
```

Si partimos de un Docker recién instalado o hacemos un `prune` de imágenes y contenedores, deberíamos encontrar esas carpetas vacías.

> El comando `prune` se utiliza para eliminar las imágenes o contenedores no utilizados.

Ahora haremos un `pull` de una imagen:

```
# docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
c9b1b535fdd9: Pull complete
Digest:
sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367
d
```

```
Status: Downloaded newer image for alpine:latest

# docker images
REPOSITORY TAG         IMAGE ID          CREATED
SIZE
alpinelatest           e7d92cdc71fe      18 hours ago
5.59MB

# ls imagedb/metadata/sha256
<Vacío - no tenemos imágenes locales>

# ls imagedb/content/sha256/
e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a
```

Como se puede observar, la carpeta *metadata* sigue vacía, porque se utiliza únicamente cuando se construyen imágenes locales. También podemos observar que hay una capa en `imagedb/content/sha256`.

Sin embargo hay algo que no cuadra: hemos descargado la imagen `ab00606a4262` pero en la carpeta vemos `ab00606a4262` y `e7d92cdc71fe` respectivamente.

¿Qué ocurre?

La explicación es que `ab00606a4262` es el *digest* del manifest, y `e7d92cdc71fe` es el *digest* de la configuración de la imagen:

> Como vimos en el punto el *digest* es uno de los componentes de la especificación de una imagen (correspondiente con el ID de la misma), junto con la configuración de la imagen y la serialización de los sistemas de archivos .

```
# cat repositories.json  | jq
{
  "Repositories": {
    "alpine": {
      "alpine:latest":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
6a",
      "alpine@sha256:

ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
6a"
    }
  }
}
```

`Repositories.json` registra los distintos repositorios conocidos por Docker. Además, para cada `repositorio:tag` , o bien, `repositorio@sha256:`*`digest_del_manifest`* (dos formatos de repositorio diferentes), registra la relación con el *digest* de la configuración de la imagen, que vemos a continuación:

```
# cat
imagedb/content/sha256/e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e
8e95d75ca6a99776a |
jq
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
      "/bin/sh"
    ],
    "ArgsEscaped": true,
    "Image":
"sha256:c80f3a794aba97667bd449f9cc59b2c72d593a8a3ca170b65ab4e57c4055e2
9f",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": null
  },
  "container":
"3e0860fab68af5a0364d739e6c851c1a1c70fc49e7c9f4f974fb79b27a19e651",
  "container_config": {
    "Hostname": "3e0860fab68a",
    "Domainname": "",
    "User": "",
```

```
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
      "/bin/sh",
      "-c",
      "#(nop) ",
      "CMD [\"/bin/sh\"]"
    ],
    "ArgsEscaped": true,
    "Image":
"sha256:c80f3a794aba97667bd449f9cc59b2c72d593a8a3ca170b65ab4e57c4055e2
9f",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": {}
  },
  "created": "2020-01-18T01:19:37.187497623Z",
  "docker_version": "18.06.1-ce",
  "history": [
    {
      "created": "2020-01-18T01:19:37.02673981Z",
      "created_by": "/bin/sh -c #(nop) ADD
file:e69d441d729412d24675dcd33e04580885df99981cec43de8c9b24015313ff8e
in / "
    },
    {
      "created": "2020-01-18T01:19:37.187497623Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
      "empty_layer": true
    }
  ],
  "os": "linux",
  "rootfs": {
    "type": "layers",
    "diff_ids": [
     "sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02a
ecceb10"
```

```
    ]
  }
}
```

Este es el fichero que nos va a permitir crear un contenedor a partir de la imagen. En la parte final vemos la composición del `rootfs`, en este caso con una única capa. Nos lista los `diff_ids` de cada capa, que son el *digest* del contenido de esta. Pero nuevamente vemos aquí algo que no cuadra:

Al principio del ejercicio, hemos descargado con el `pull` la capa **`c9b1b535fdd9`**, pero en el archivo de configuración nos aparece **`5216338b40a7`**.

¿Qué ocurre?

La explicación es que al hacer el `pull`, el *digest* corresponde a la capa **comprimida**, mientras que localmente, los *digests* corresponden al contenido **descomprimido**. La compresión se realiza para hacer más eficiente la transferencia, pero una vez descargado se descomprime. Por eso el fichero de configuración de la imagen indica los *digests* de las capas descomprimidas. Podemos ver la relación aquí:

```
# cat distribution/diffid-by-digest/sha256/ c9b1b535fdd91a9855fb7f8
2348177e5f019329a58c53c47272962dd60f71fc9
Sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb1
0

# cat
distribution/v2metadata-by-diffid/sha256/5216338b40a7b96416b8b9858974b
be4acc3096ee60acbc4dfb1ee02aecceb10
[
  {
    "Digest": "sha256:c9b1b535fdd91a9855fb7f82348177e5f019329a58c53
c47272962dd60f71fc9",
    "SourceRepository": "docker.io/library/alpine",
    "HMAC": ""
  }
]
```

El `diff_id` de una capa nos permite buscar esa capa y analizar qué guarda Docker en la carpeta `layerdb`, tal y como se observa a continuación.

Nota: En este caso realmente no es el diff_id, sino el ChainID, esto es, el *digest* de

la composición de todas las capas inferiores, pero en este caso, al ser la capa más baja, ambos (diff_id y Chain ID) coinciden.

```
# ls -lh
layerdb/sha256/5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee0
2aecceb10/
total 32K
-rw-r--r-- 1 root root  64 ene 18 20:24 cache-id
-rw-r--r-- 1 root root  71 ene 18 20:24 diff
-rw-r--r-- 1 root root   7 ene 18 20:24 size
-rw-r--r-- 1 root root 19K ene 18 20:24 tar-split.json.gz

# cat
layerdb/sha256/5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee0
2aecceb10/diff
sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb1
0

# cat layerdb/sha256/5216338b40a7b96416b8b9858974bbe4acc3096ee60acb
c4dfb1ee02aecceb10/cache-id
3f7664a0535e183d785b884b2827fa3d4cd6f63fc3379dc003e8c2467c69dcdd
```

El archivo `diff` contiene el DiffID correspondiente a este ChainID (que en este caso es el mismo). El `cache-id` es el identificador de la carpeta, aleatoriamente generado, donde guardamos todo lo referente a esta capa (lo que encontremos ya dependerá del driver de almacenamiento)

Sin entrar en detalles, el archivo `tar-split.json.gz` guarda cabeceras del archivo en formato `tar` para poder reconstruirlo exactamente igual sin que cambie el *digest* a partir de los contenidos de la imagen.

Para ampliar la información sobre estos archivos, consúltese:

https://github.com/vbatts/tar-split

A continuación vamos al directorio `/var/lib/docker/overlay2`, que es donde se almacena realmente el contenido de las capas (dependiendo del driver utilizado, claro):

```
# cd /var/lib/docker/overlay2

# ls
3f7664a0535e183d785b884b2827fa3d4cd6f63fc3379dc003e8c2467c69dcdd  l

# ls -l l
total 4
lrwxrwxrwx 1 root root 72 ene 18 20:43 6FSF2G42R54W7OIPXLXMBNLDUQ -
> ..
/3f7664a0535e183d785b884b2827fa3d4cd6f63fc3379dc003e8c2467c69dcdd/diff

# ls
3f7664a0535e183d785b884b2827fa3d4cd6f63fc3379dc003e8c2467c69dcdd/diff/
bin  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv
sys  tmp  usr  va
```

La carpeta especial «`l`» minúscula, tal y como se indica en la documentación que veremos más adelante, se usa para acortar los nombres y evitar problemas con `mount`.

En la carpeta `diff` encontramos el contenido del sistema de archivos del contenedor.

La misma información que hemos visto examinando el archivo de configuración de la imagen, la podemos obtener mediante el comando `docker inspect`, como se muestra a continuación:

```
# docker inspect alpine
[
    {
        "Id": "sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d2
9e8e95d75ca6a99776a",
        "RepoTags": [
            "alpine:latest"
        ],
        "RepoDigests": [
            "alpine@sha256:ab00606a42621fb68f2ed6ad3c88be54397f981
a7b70a79db3d1172b11c4367d"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2020-01-18T01:19:37.187497623Z",
        "Container": "3e0860fab68af5a0364d739e6c851c1a1c70fc49e7c9f
4f974fb79b27a19e651",
```

```
        "ContainerConfig": {
            "Hostname": "3e0860fab68a",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
                       usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/sh\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:c80f3a794aba97667bd449f9cc59b2c72d59
3a8a3ca170b65ab4e57c4055e29f",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "18.06.1-ce",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/
                       usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh"
```

```
            ],
            "ArgsEscaped": true,
            "Image": "sha256:c80f3a794aba97667bd449f9cc59b2c72d59
3a8a3ca170b65ab4e57c4055e29f",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 5591300,
        "VirtualSize": 5591300,
        "GraphDriver": {
            "Data": {
                "MergedDir":
"/var/lib/docker/overlay2/3f7664a0535e183d785b884b2827fa3d4cd6f63fc337
9dc003e8c2467c69dcdd/merged",
                "UpperDir":
"/var/lib/docker/overlay2/3f7664a0535e183d785b884b2827fa3d4cd6f63fc337
9dc003e8c2467c69dcdd/diff",
                "WorkDir":
"/var/lib/docker/overlay2/3f7664a0535e183d785b884b2827fa3d4cd6f63fc337
9dc003e8c2467c69dcdd/work"
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee6
0acbc4dfb1ee02aecceb10"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        }
    }
]
```


Si todo se ha ejecutado correctamente, esto debería corresponder con lo que hay almacenado en:

`imagedb/content/sha256/e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a`

Ejercicio 2: qué ocurre cuando creamos nuestra propia imagen

> Sugerencia para la resolución del ejercicio: ir anotando aparte los diferentes IDs para facilitar su seguimiento durante la resolución del ejercicio y para comparar con la imagen del ejercicio anterior.

A continuación, vamos a construir una imagen propia basada en Alpine (que usa como base la distribución Alpine Linux[9]) y analizaremos los archivos y carpetas que se generan en `/var/lib/docker`:

```
# cat Dockerfile
FROM alpine:latest
ADD test.txt /

# docker build -t myimage .
Sending build context to Docker daemon  3.072kB
Step 1/2 : FROM alpine:latest
 ---> e7d92cdc71fe
Step 2/2 : ADD test.txt /
 ---> 09c0177179d4
Successfully built 09c0177179d4
Successfully tagged myimage:latest

# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
myimage             latest              09c0177179d4        44 seconds ago
5.59MB
alpine              latest              e7d92cdc71fe        19 hours ago
5.59MB

# # cat /var/lib/docker/image/overlay2/repositories.json  | jq
{
  "Repositories": {
    "alpine": {
```

9 https://www.alpinelinux.org/

```
      "alpine:latest":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
6a",

"alpine@sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172
b11c4367d":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
6a"
    },
    "myimage": {
      "myimage:latest":
"sha256:09c0177179d4931cdd51963086f07792928a179784a4e0ec86398906141e8f
ab"
    }
  }
}
```

Podemos apreciar que como la imagen aún no tiene un manifest, ya que no hemos hecho `push` en ningún registro, no existe una entrada para `myimage` con el *digest* del manifest en el formato `myimage@sha256:xxxx`

Veamos entonces el fichero de configuración generado:

```
# cat
/var/lib/docker/image/overlay2/imagedb/content/sha256/09c0177179d4931c
dd51963086f07792928a179784a4e0ec86398906141e8fab  | jq
{
  "architecture": "amd64",
  "config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
```

```
      "/bin/sh"
    ],
    "ArgsEscaped": true,
    "Image": "sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8
e95d75ca6a99776a",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": null
  },
  "container_config": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": [
      "/bin/sh",
      "-c",
      "#(nop) ADD
file:720fdd903fdd610433e433e6095dfd5cbcf64baae4985a7c281a311b1d48a512
in / "
    ],
    "ArgsEscaped": true,
    "Image": "sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e
8e95d75ca6a99776a",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": null
  },
  "created": "2020-01-18T19:49:54.06082304Z",
  "docker_version": "18.09.7",
  "history": [
    {
      "created": "2020-01-18T01:19:37.02673981Z",
      "created_by": "/bin/sh -c #(nop) ADD
```

```
file:e69d441d729412d24675dcd33e04580885df99981cec43de8c9b24015313ff8e
in / "
    },
    {
      "created": "2020-01-18T01:19:37.187497623Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
      "empty_layer": true
    },
    {
      "created": "2020-01-18T19:49:54.060823042",
      "created_by": "/bin/sh -c #(nop) ADD
file:720fdd903fdd610433e433e6095dfd5cbcf64baae4985a7c281a311b1d48a512
in / "
    }
  ],
  "os": "linux",
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee0
2aecceb10",
      "sha256:e9f56d359a248e157649654e87fac66c376f4d8c6ba6af633ff1f1c
49caa2cde"
    ]
  }
}
```

El fichero de configuración nos indica que se ha creado sobre la imagen: «**e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a**» de Alpine, añadiendo un fichero.

Además, comparte la capa: «**5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10**» con Alpine, y añade la capa: «**e9f56d359a248e157649654e87fac66c376f4d8c6ba6af633ff1f1c49caa2cde**»

```
# ls /var/lib/docker/image/overlay2/imagedb/metadata/sha256/
09c0177179d4931cdd51963086f07792928a179784a4e0ec86398906141e8fab
```

Ahora la carpeta “metadata” no está vacía, puesto que esta vez sí se ha construido una imagen local. Contiene información como *lastUpdated* y la imagen padre (*parent)*.

```
# ls /var/lib/docker/image/overlay2/layerdb/sha256/
1db6b94dcd26f48dc930a2d42afb3f079c86558cfdf4411914e4e91dff0f0f52
5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10
```

Esperábamos encontrar aquí una carpeta con el nombre: «`e9f56d359a248e157649654e87fac66c376f4d8c6ba6af633ff1f1c49caa2cde`» para contener la información de la capa que añadimos en nuestro contenedor, pero hay algo que no cuadra ya que esa carpeta no está ahí.

¿Qué ocurre?

La explicación es que, como hemos comentado previamente sin entrar en detalles, el nombre de los directorios en:

`/var/lib/docker/image/overlay2/layerdb/sha256/`

no son los *diff_id* (que es el *digest* del diff, es decir, la capa que contiene sólo las diferencias), sino que se trata del *chainID*, que es el *digest* de la capa completa una vez compuesta por sus ancestros.

El *chainID* de la capa más baja es igual que el *diff_id*, pero no es igual al del resto de las capas que no estén vacías. Amplíese la información sobre esto en el siguiente enlace:

https://github.com/opencontainers/image-spec/blob/master/config.md#layer-chainid

Sin embargo, podemos calcular el ChainID, comprobar la información almacenada y, además, cómo el archivo *diff* sí que contiene el DiffID que esperamos, también podemos localizar la carpeta de `overlay2` donde ver los contenidos:

```
#  echo -n "sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb
1ee02aecceb10
sha256:e9f56d359a248e157649654e87fac66c376f4d8c6ba6af633ff1f1c49caa2cd
e"| sha256sum -
1db6b94dcd26f48dc930a2d42afb3f079c86558cfdf4411914e4e91dff0f0f52  -

# cat
/var/lib/docker/image/overlay2/layerdb/sha256/1db6b94dcd26f48dc930a2d4
2afb3f079c86558cfdf4411914e4e91dff0f0f52/diff
```

```
sha256:e9f56d359a248e157649654e87fac66c376f4d8c6ba6af633ff1f1c49caa2cd
e

# cat
/var/lib/docker/image/overlay2/layerdb/sha256/1db6b94dcd26f48dc930a2d4
2afb3f079c86558cfdf4411914e4e91dff0f0f52/cache-id
d3074c8909f5c6fe23ed986d08fe254c32666b6db26ab5ada2aad4791715c5ae

# ls
/var/lib/docker/overlay2/d3074c8909f5c6fe23ed986d08fe254c32666b6db26ab
5ada2aad4791715c5ae/diff/
test.txt
```

Haremos ahora un `push` de la imagen. Tendremos que cambiar el tag para poder hacer `push` a un repositorio personal y analizar nuevamente qué ocurre con los *digests* al hacer `push` y `pull` en un ordenador distinto:

```
# docker push myimage
The push refers to repository [docker.io/library/myimage]
e9f56d359a24: Preparing
5216338b40a7: Preparing
denied: requested access to the resource is denied

# docker tag myimage airadier/test
# docker push airadier/test
The push refers to repository [docker.io/airadier/test]
e9f56d359a24: Pushed
5216338b40a7: Mounted from library/alpine
latest: digest:
sha256:8360f9950adc9101a502359aa6ccda4c28652b2e90afb3c34cd78b5330638ff
e size: 735

# docker images --digests
REPOSITORY          TAG                 DIGEST
IMAGE ID            CREATED             SIZE

airadier/test       latest
sha256:8360f9950adc9101a502359aa6ccda4c28652b2e90afb3c34cd78b5330638ff
e   09c0177179d4    About an hour ago   5.59MB

myimage             latest              <none>
09c0177179d4        About an hour ago   5.59MB

alpine              latest
sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367
```

```
d   e7d92cdc71fe 20 hours ago            5.59MB
```

Ahora cambiamos a un ordenador distinto:

```
# $ docker pull airadier/test
Using default tag: latest
latest: Pulling from airadier/test
c9b1b535fdd9: Pull complete
cd84a87f18fe: Pull complete
Digest: sha256:8360f9950adc9101a502359aa6ccda4c28652b2e90afb3c34cd
78b5330638ffe
Status: Downloaded newer image for airadier/test:latest
docker.io/airadier/test:latest
```

Podemos apreciar que el ID de la imagen (indicado en la columna «`IMAGE ID`») es el *digest* del fichero de configuración de la imagen, mientras que el *digest* (que aparece al añadir "`--digests`", y que no está disponible mientras nuestra imagen exista sólo localmente) es el *digest* del fichero manifest, que es generado por el registro (y no localmente). Por eso la imagen local, no *pusheada*, no tiene *digest*.

Además vemos que es distinto el *digest* de las capas al hacer `pull` y al hacer `push`, ya que en el `pull` vemos el *digest* de la imagen comprimida (tal y como la estamos descargando del repositorio) mientras que al hacer `push`, nos está mostrando el DiffID tal cual, sin comprimir.

Recopilando...

A continuación se muestra una tabla resumen (Tabla 3) que podemos ir rellenando conforme avancemos con la resolución del ejercicio, para facilitar el seguimiento de cada *digest*: Posteriormente se comenta cada uno de estos conceptos.

Tabla 3: Tabla de ayuda para la resolución de ejercicios

Comando	alpine	myimage
Imagen pulled (manifest)	ab00606a4262	8360f9950adc
IMAGE ID (config)	e7d92cdc71fe	09c0177179d4
Image dentro de config	c80f3a794aba	e7d92cdc71fe
Capa pull / push - 0	c9b1b535fdd9	c9b1b535fdd9 / 5216338b40a7
Diff ID - 0	5216338b40a7	5216338b40a7
Chain ID en layerdb - 0	5216338b40a7	5216338b40a7
Cache ID - 0	3f7664a0535e	3f7664a0535e
Capa pull / push - 1	-	cd84a87f18fe / e9f56-d359a24
Diff ID - 1	-	e9f56d359a24

Comando	alpine	myimage
`Chain ID en layerdb - 1`	-	`1db6b94dcd26`=hash(`5216338b40a7`|`e9f56-d359a24`)
`Cache ID - 1`	-	`d3074c8909f5`

- **Manifest**: El *digest* del manifest de la imagen, que contiene el *digest* de la configuración de la imagen y las capas, con el *digest* (DiffID) de cada capa.
- **DiffID**: El *digest* sobre el archivo `tar` descomprimido y serializado. Las capas se desempaquetan y empaquetan de forma reproducible para evitar que cambie el DiffID, usando `tar-split` para guardar las cabeceras de `tar`.
- ***ImageID***: El *digest* del fichero de configuración de la imagen, que contiene datos de configuración y la lista de capas. Es necesario para poder crear un contenedor.
- ***Image dentro del fichero config:*** podría parecer que indica la imagen sobre la que se construyó, como es el caso de nuestra imagen, ya que aparece el ImageID de Alpine. Sin embargo, realmente es el ImageID de la última orden en el Dockerfile. Ya sabemos que en cada paso de Dockerfile, Docker va generando una imagen intermedia con el resultado de aplicar cada comando. Como nuestra imagen ejemplo sólo tiene un paso, y se aplica sobre Alpine, coincide con el ImageID de éste. Si hubiera más pasos en el Dockerfile, este *digest* sería de una de las imágenes intermedias.
- ***ChainID:*** es el *digest* pero de la composición de todas las capas inferiores. Por lo tanto, es el *digest* sobre el contenido completo de todas las capas unificadas hasta ese nivel (a diferencia del DiffID, que es el *digest* únicamente sobre el contenido de esa capa). Para ampliar la

información consulten el enlace:

https://github.com/opencontainers/image-spec/blob/master/config.md#layer-chainid

- ***CacheID:*** es un identificador aleatorio, generado durante la descarga o durante el *build*, para almacenar un ChainID concreto.

¿Y el manifest dónde se guarda?

El contenido del manifest es simplemente el "`sha256`" y el *media type* del fichero de configuración, y las layers. Se usa simplemente para descargar el config de la imagen y las capas. Una vez se ha descargado la imagen, el manifest ya no tiene sentido, y no se usa para la gestión local de las imágenes, así que no se almacena localmente (sólo existe en el *registro*).

Referencias adicionales

En las siguientes páginas web se puede encontrar información complementaria que nos ayudará a seguir profundizando a cerca de todo lo visto en los apartados anteriores:

- Walk into docker(05): How does docker manage image s locally? [enlace]

  https://programmer.help/blogs/walk-into-docker-05-how-does-docker-manage-image-s-locally.html

- Explaining Docker Image Ids [enlace]

  https://windsock.io/explaining-docker-image-ids/

- A Peek into Docker Images - Tenable TechBlog [enlace]

  https://medium.com/tenable-techblog/a-peek-into-docker-images-b4d6b2362eb

Networking

El **networking** en Docker, son las diferentes posibilidades de configuración de la red que permiten que todos los contenedores aislados se comuniquen entre sí y con el Docker host en diversas situaciones para realizar las acciones requeridas, permitiendo crear aplicaciones que funcionan juntas de forma segura. [46] [47]

Los contenedores pueden comunicarse entre ellos a través de varios métodos. En este caso vamos a explicar las diferentes opciones de configuración:

- None
- Bridge
- Host
- MacVlan
- Overlay
- Third Party Plugins

None

Básicamente, `<none>` indica que el contenedor no tiene una interfaz de red como tal, aunque si dispone de una interfaz de loopback (esto es, virtual). Este modo se utiliza en contenedores que no usan la red, como pueden ser contenedores para pruebas que no necesitan comunicarse con nada, o aquellos que se pueden dejar preparados para conectarse posteriormente a una red.

> La interfaz de red virtual loopback se suele utilizar cuando una transmisión de datos tiene como destino el propio host. También se suele usar en tareas de diagnóstico de conectividad y validez del protocolo de comunicación.[48]

Más información [enlace]:

https://docs.docker.com/network/none/

IPTables y modo bridge

El modo **bridge** es la configuración de red que toman por defecto los contenedores de Docker.

Docker usa dispositivos de red virtuales `vethX` (virtual ethernet), y crea *switches virtuales* (que permiten la comunicación entre ellos) en el kernel Linux mediante "*bridges*", un módulo introducido sobre el año 2000 en el kernel 2.2. que proporciona una red interna en el host a través de la cual pueden comunicarse los contenedores.

Los bridges se pueden gestionar con la utilidad **`brctl`** (que permite configurar, mantener e inspeccionar la configuración del bridge), y posibilita la conexión de múltiples interfaces como si estuvieran físicamente enlazados en un switch.

Un ejemplo típico de utilización de `brctl` suele ser su uso en los puntos de acceso o routers wifi con un sistema operativo Linux, para conectar el interfaz `wlan0` y `eth0`.

Las direcciones de la red interna del host no son accesibles desde fuera del host y se aprovechan de IPtables (en el caso del protocolo IPv4[10]) para realizar el mapeado de puertos y el nateo, esto es, hacer NAT (Network Address Translation), redirigiendo las peticiones externas a un puerto hacia un ordenador determinado. Así, mediante reglas de iptables, Docker consigue el aislamiento entre contenedores. Las reglas se crean mediante una *chain* de nombre `DOCKER`, que no debería ser manipulada manualmente. Si se quieren añadir reglas adicionales, como por ejemplo, restringir el acceso por dirección IP al Docker daemon, estas se deben añadir a un *chain* de nombre `DOCKER-USER`.

> IPTables es una herramienta de Linux que nos permite implementar firewalls, filtrando o restringiendo el tráfico de red en un sistema, tanto entrante como saliente, a través de tablas con reglas de filtrado de paquetes Ipv4. [49]

10 https://en.wikipedia.org/wiki/IPv4

La creación de un contenedor con la configuración de red en modo bridge incluiría, entonces, los siguientes pasos:

- Se proporcionará una red bridge al host
- En cada bridge se proporcionará un namespace diferente
- Las interfaces de red de los contenedores serán mapeadas a las interfaces privadas del bridge
- Se realizarán "nateos" en `iptables` para mapear cada contenedor con la interfaz pública/privada del host.

Al exponer un puerto en un contenedor, se crean una serie de reglas de iptables, el bridge, y un proceso docker-proxy que escucha en el puerto expuesto (lo que puede consumir memoria en caso de que sean muchos puertos los que se expongan).

> El proceso docker-proxy se usa para manejar conexiones que se originan en la máquina local que, de otro modo, no pasarían por las reglas de iptables, o cuando Docker se ha configurado de manera que no manipule iptables en absoluto.[50]

Desde la versión 1.7.0 se puede deshabilitar este proceso (usando la opción `--userland-proxy=false`), y usar la funcionalidad llamada hairpin NAT, que realiza la redirección al contenedor usando exclusivamente reglas de NAT con iptables. Aunque esto tiene la desventaja de que si ya existe un proceso escuchando en el host, los puertos colisionan, y la regla de NAT de iptables toma precedencia, enmascarando al servicio ya existente.

```
# docker build -t envtest - << EOF
FROM alpine:latest

RUN apk --update add bind-tools && rm -rf /var/cache/apk/*
EXPOSE 80
EOF
```

Creamos una red de prueba y un par de contenedores en esa red (`-P` expone todos los puertos que están definidos con `EXPOSE`, usando puertos aleatorios):

```
# docker network create test
```

```
$ brctl show
bridge namebridge id               STP enabled interfaces
br-14af8973e9cc  8000.02420f5c1fc6  no
docker0          8000.0242886b9be6  no


# docker run --net test -dit --name host1 -P envtest sh
# docker run --net test -dit --name host2 -P envtest sh
# docker ps

$ docker ps
CONTAINER ID            IMAGE              COMMAND      CREATED
STATUS                  PORTS                           NAMES

eedc026229c5            envtest            "sh"         4 minutes ago
Up 4 minutes            0.0.0.0:32769->80/tcp           host2
2a31d4a6a0ae            envtest            "sh"         4 minutes ago
Up 4 minutes            0.0.0.0:32768→80/tcp            host1

$ brctl show
bridge name             bridge id               STP enabled
interfaces

br-14af8973e9cc         8000.02420f5c1fc6       no
veth0816cc6       vethb462342

docker0                 8000.0242886b9be6       no

# iptables -nvL

…
Chain DOCKER (2 references)
 pkts      bytes target      prot opt in
out              source              destination

 0         0     ACCEPT      tcp  --  !br-41c024439415 br-41c024439415
0.0.0.0/0        172.18.0.2          tcp dpt:80

 0         0     ACCEPT      tcp  --  !br-41c024439415 br-41c024439415
0.0.0.0/0        172.18.0.3          tcp dpt:80
...
```

Nos fijamos que en la `chain DOCKER` hay un `ACCEPT` para los puertos 80 en dos contenedores. Además, al pertenecer los contenedores a la misma red, sus interfaces se añaden al mismo bridge, lo que podemos verificar con la

herramienta **brctl**.

Un puerto se abre por cada puerto expuesto en la imagen del contenedor:

```
# ss -tan | grep LISTEN
LISTEN           0           128         *:22        *:*
LISTEN           0           128         :::22       :::*
LISTEN           0           128         :::32770    :::*
LISTEN           0           128         :::32771    :::*
```

Con la configuración por defecto, vemos el proceso `docker-proxy`, de la siguiente manera:

```
# ps -Af | grep proxy

root       10010 4219  0 17:46 ?    00:00:00 /usr/bin/docker-proxy -
proto tcp -host-ip 0.0.0.0 -host-port 32770 -container-ip 172.18.0.2 -
container-port 80

root       10157 4219  0 17:46 ?    00:00:00 /usr/bin/docker-proxy -
proto tcp -host-ip 0.0.0.0 -host-port 32771 -container-ip 172.18.0.3 -
container-port 80
```

Como vemos en el siguiente código, si exponemos varios puertos, vemos que hay un muchos procesos proxy: [¡Ojo porque estos consumen memoria!]:

```
# docker run --net test -dit --name prangetest -p 76-85:76-85 envtest
sh

# ps -o pid,%cpu,%mem,sz,vsz,cmd -A --sort -%mem | grep proxy

10388  0.0  0.0 74718 298872 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 81 -container-ip 172.18.0.4 -container-port 81

10427  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 78 -container-ip 172.18.0.4 -container-port 78

10010  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 32770 -container-ip 172.18.0.2 -container-port 80

10157  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 32771 -container-ip 172.18.0.3 -container-port 80
```

```
10440  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 77 -container-ip 172.18.0.4 -container-port 77

10376  0.0  0.0 54236 216944 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 82 -container-ip 172.18.0.4 -container-port 82

10402  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 80 -container-ip 172.18.0.4 -container-port 80

10415  0.0  0.0 54588 218352 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 79 -container-ip 172.18.0.4 -container-port 79

10453  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 76 -container-ip 172.18.0.4 -container-port 76

10351  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 84 -container-ip 172.18.0.4 -container-port 84

10324  0.0  0.0 56285 225140 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 85 -container-ip 172.18.0.4 -container-port 85

10364  0.0  0.0 54588 218352 /usr/bin/docker-proxy -proto tcp -host-ip
0.0.0.0 -host-port 83 -container-ip 172.18.0.4 -container-port 83

10542  0.0  0.0  3235  12940 grep --color=auto proxy
```

Por último, no olvidemos parar los contenedores, de la siguiente manera:

```
$ docker stop host1 -t 0
$ docker stop host2 -t 0
$ docker stop prangetest -t 0
$ docker container rm ...
$ docker container prune
```

Con respecto a la **resolución de nombres** en la red bridge, es importante saber que, si bien Docker configura una resolución DNS (Domain Name System) para poder resolver los nombres de otros contenedores, en la red bridge, por defecto, no funciona la resolución de nombres de contenedores: solo está permitida la comunicación entre ellos a través de direcciones IP. Si se crea otra red bridge, los contenedores tendrán comunicación entre ellos, y además la configuración DNS permitirán resolver el nombre del contenedor por su IP.

Resolución DNS, o resolución de nombres, es el proceso de traducción de los nombres de domi-

nio a direcciones numéricas que las máquinas puedan entender [51]

También pueden crearse nuevas **redes definidas por el usuario** para crear una red de tipo bridge:

- Las redes definidas por usuario se crean con `docker network create`, y cada red crea un nuevo bridge.
- Las redes de usuario permiten un mejor aislamiento e interoperabilidad entre aplicaciones. Los contenedores conectados a la misma red automáticamente exponen todos sus puertos entre ellos, pero ninguno al exterior. Por ejemplo, en el caso de una base de datos y una aplicación usada en la misma red.
- Las redes de usuario proveen resolución DNS entre contenedores. En otro caso, es necesario usar la opción `–link`, pese a que esta se considere "*legacy*" (heredada).
- Se pueden añadir y quitar contenedores ``al vuelo´´ con el comando `docker network connect.` Para la red por defecto, hay que detenerlos.

Las opciones de la red bridge por defecto se pueden cambiar en el fichero de configuración de dockerd daemon.json

➔ No se recomienda usar la red bridge para producción.

Véanse los ejemplos que se muestran en la siguiente página web:

https://docs.docker.com/network/network-tutorial-standalone/

Y amplíese la información a cerca de la red bridge en esta otra:

https://docs.docker.com/network/bridge/

Modo host

En la **configuración host**, simplemente **se elimina el aislamiento de red** y se

utilizan los mismos interfaces que el host (el contenedor tendrá acceso a todas las interfaces de red del host). Ahora, el contenedor comparte su namespace de red con el del host, lo que dará lugar a un mejor rendimiento, ya que elimina la necesidad del uso de NAT (enmascaramiento de IP) Esto puede ser bueno en ciertos escenarios, como cuando hay muchísimos puertos expuestos por un contenedor, o en escenarios de auto-descubrimiento (*clusters* del software Hazelcast), en los que un nodo del cluster intenta descubrir a otro.

El hecho de que la IP a la que lanza el paquete de descubrimiento no coincida con la IP que internamente reporta el contenedor descubierto, puede causar problemas (en versiones nuevas de Hazelcast se contemplan estos escenarios, y existen opciones de configuración específicas).

Las opciones de exponer/publicar puertos se ignoran, ya que no tienen sentido en este modo de configuración.

Más información en este enlace:

https://docs.docker.com/network/host/

Modo macvlan

La configuración de red `MacVlan` **permite asignar una dirección** MAC (Media Access Control, o "dirección física") **a cada contenedor**, haciendo que aparezcan como máquinas físicas independientes en la red. El *demonio* (o daemon*;* ambas palabras se usan indistintamente) de Docker enruta el tráfico a los contenedores a través de su dirección MAC.

Es necesario desasignar una interfaz física del host para poder utilizarla en el contenedor.

En este modo de red, el interfaz virtual del contenedor funciona como si estuviera conectado físicamente a la red especificada por el adaptador padre (al crear la red con `docker network create` se especifica con el parámetro `parent`, como se observa en el ejemplo de la Figura 24).

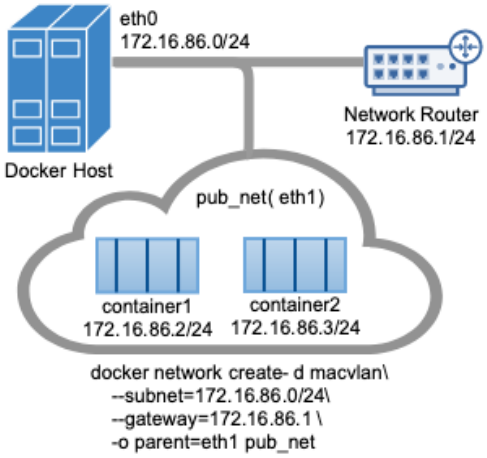

Figura 24: Ejemplo simple de modo macvlan

El modo `macvlan` es útil para ciertas herramientas, como monitores de tráfico de red, que requieren un acceso físico (Layer 2) a la red.

Tiene algunos inconvenientes, como que por defecto no es posible que la dirección IP asignada al contenedor se obtenga por protocolo DHCP de la propia red, sino que la asigna Docker a partir de los parámetros de *`subnet`* utilizados al crear la red.

Existe un driver experimental: *LibNetwork*, pero este requiere modificar y re-compilar algunos componentes de Docker. Para más información a cerca de esto consulte este enlace:

https://gist.github.com/nerdalert/3d2b891d41e0fa8d688c

Y para más información sobre este modo de red:

- Use macvlan networks

  https://docs.docker.com/network/macvlan/

- Macvlan and IPvlan basics

https://sreeninet.wordpress.com/2016/05/29/macvlan-and-ipvlan/

- Macvlan vs Ipvlan

http://hicu.be/macvlan-vs-ipvlan

Redes Overlay

Las redes ***overlay*** son redes "virtuales" y distribuidas que se provee de túneles de red para interconectar distintos daemons Docker y permitir la comunicación entre contenedores situados en en distintos hosts de forma transparente, como si pertenecieran a la misma red física. Así, la red overlay se encarga del direccionamiento y enrutado de forma transparente para las aplicaciones, y se utiliza principalmente en la orquestación de contenedores multi-host. Para utilizar estas redes multi-host se requieren algunos parámetros adicionales al lanzar el *demonio* (daemon) de Docker, así como un registro clave-valor, es decir, un servicio centralizado adicional que permita a Docker almacenar y recuperar estos registros con la correspondencia entre los servicios de la red y sus direcciones correspondientes. Usando Docker Swarm la mayoría de esta configuración es transparente, pero al final del capítulo veremos un ejercicio en el que configuramos una red Overlay de forma manual, usando Consul como servicio de registr.

Docker utiliza la tecnología VXLAN (Virtual Extensible Local Area Network) incluida de forma nativa desde la versión 1.9.

Las redes overlay se utilizan especialmente con la herramienta Docker Swarm y con otros orquestadores, como Kubernetes. Algunos ejemplos de redes overlay, además de la propia de Docker Swarm, son Flannel, Calico, Cilium, Contiv, Canal, Weave, etc.

https://docs.docker.com/network/overlay/

Third Party Plugins

Además de los tipos de redes explicados anteriormente, es posible la creación de

plugins propios o, el uso, de plugins existentes creados por otras empresas, también llamados plugins de terceros, para llevar a cabo la configuración de ciertas funcionalidades de forma sencilla o más personalizada.

Ejercicio 3: crear una red overlay para acceder a un servicio en otra máquina

Vamos a crear un ejemplo de red Overlay sin desplegar Docker Swarm, simplemente usando Consul[11], una herramienta para descubrimiento de servicios.

Primero, crearemos la máquina para Consul:

```
$ docker-machine create -d virtualbox keyvalue
```

Ahora podemos entrar en la máquina con `docker-machine ssh keyvalue`, o bien ejecutar con el cliente local:

```
$ docker (docker-machine config keyvalue) pull progrium/consul

$ docker-machine ssh keyvalue

docker@keyvalue:~$ docker run -d -p 8500:8500 progrium/consul -server
-bootstrap
5620a5efc7e4e524a18c5e05935696e177a1883f27865c0fee9ec845f0b321cd
docker@keyvalue:~$ exit

$ docker-machine ip keyvalue
192.168.99.100
```

Si abrimos en el navegador http://192.168.99.100:8500 veremos el panel de Consul [Figura 25]:

11 Consul: https://github.com/hashicorp/consul

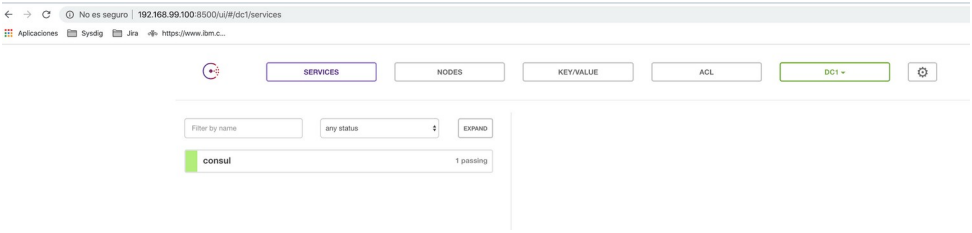

Figura 25: Panel de Consul

Ahora crearemos dos máquinas ejecutando Docker, conectadas a Consul, y una red `overlay` en la máquina `node-0`:

```
$ docker-machine create -d virtualbox \
 --engine-opt="cluster-store=consul://$(docker-machine ip keyvalue):8500" \
 --engine-opt="cluster-advertise=eth1:2376" \
 node-0
…

$ docker-machine create -d virtualbox \
 --engine-opt="cluster-store=consul://$(docker-machine ip keyvalue):8500" \
 --engine-opt="cluster-advertise=eth1:2376" \
 node-1
…

$ docker-machine ssh node-0
…
docker@node-0:~$ docker network create -d overlay multi-host-net
4fab78411c110e9fa3ba1af1424a1ed1a96710e235b21ea23220e87596f51bcd
```

A continuación, listaremos las redes disponibles en la máquina `node-1`:

```
$ docker-machine ssh node-1
…
docker@node-1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
e34263bdcf9c        bridge              bridge              local
24f6d34448dc        host                host                local
4fab78411c11        multi-host-net      overlay             global
59849a509e4d        none                null                local
```

Y “mágicamente” (gracias a Consul y a las opciones configuradas en el daemon Docker), la red creada en el nodo 0 aparece también en el nodo 1.

Ahora arrancaremos un servidor `nginx` en este mismo nodo, y a continuación probaremos a acceder a este `nginx` desde la otra máquina:

```
docker@node-1:~$ docker run -d -p80:80 --net=multi-host-net --
name=webserver nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
8ec398bc0356: Pull complete
dfb2a46f8c2c: Pull complete
b65031b6a2a5: Pull complete
Digest:
sha256:8aa7f6a9585d908a63e5e418dc5d14ae7467d2e36e1ab4f0d8f9d059a3d071c
e
Status: Downloaded newer image for nginx:latest
150dbd2b1cfd1a87ea19192be464a37b719c4559a76ec45ef3fb07c0895422e4

docker@node-1:~$ curl localhost
<!DOCTYPE html>
<html>
…
<h1>Welcome to nginx!</h1>
…
</html>

docker@node-1:~$ exit

$ docker-machine ssh node-0
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

docker@node-0:~$ docker run -ti --rm --net=multi-host-net alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
c9b1b535fdd9: Pull complete
Digest:
sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367
d
Status: Downloaded newer image for alpine:latest

/ # ping webserver
PING webserver (10.0.0.2): 56 data bytes
64 bytes from 10.0.0.2: seq=0 ttl=64 time=0.755 ms
^C
--- webserver ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.755/0.755/0.755 ms
```

```
/ # apk add curl
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86_64/APKINDEX.tar.gz
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.11/community/x86_64/APKINDEX.t
ar.gz
(1/4) Installing ca-certificates (20191127-r0)
(2/4) Installing nghttp2-libs (1.40.0-r0)
(3/4) Installing libcurl (7.67.0-r0)
(4/4) Installing curl (7.67.0-r0)
Executing busybox-1.31.1-r9.trigger
Executing ca-certificates-20191127-r0.trigger
OK: 7 MiB in 18 packages

/ # curl webserver
<!DOCTYPE html>
<html>
…
<h1>Welcome to nginx!</h1>
…
</html>
```

Para una explicación más detallada del ejercicio, consúltese:

https://codeblog.dotsandbrackets.com/multi-host-docker-network-without-swarm/

Para más información acerca de las redes overlay:

1. Use overlay networks :

   https://docs.docker.com/network/overlay/

2. Deep dive into Docker Overlay Networks : Part 1

   https://blog.d2si.io/2017/04/25/deep-dive-into-docker-overlay-networks-part-1/

3. Kubernetes networks solutions comparison

   https://www.objectif-libre.com/en/blog/2018/07/05/k8s-network-solutions-comparison/

4. Comparing Kubernetes Networking Providers

   https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

Cuestiones Prácticas

¿Qué ocurre cuando ejecutamos docker create?

Ejecutamos `docker create`:

```
$ docker create alpine
571a9e85e092fe77d52fa9fa014b94777ebbf7f4fc954d6a844768e958dfd505
$ docker ps -a
CONTAINER ID           IMAGE       COMMAND            CREATED
STATUS                 PORTS       NAMES
571a9e85e092           alpine"/bin/sh"          32 seconds ago
Created                peaceful_wescoff

/var/lib/docker/image/overlay2/layerdb/mounts# ls 571a9e85...
init-id  mount-id  parent

# cat init-id
d5cb3c905cd0fc46c4552e3e38a967d74fd5d9357de1ac719491b4da2e4e709a-init

# cat mount-id
d5cb3c905cd0fc46c4552e3e38a967d74fd5d9357de1ac719491b4da2e4e709a

# cat parent
sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb1
0
```

Como podemos observar, al crear un contenedor, Docker crea dos capas adicionales:

- La capa `init-id`, que es una capa de sólo lectura creada para añadir algunos ficheros adicionales requeridos por el contenedor (por ejemplo `/etc/resolv.conf`).
- La capa `mount-id`, que es la capa de lectura y escritura del contenedor.

Estas dos capas existen en `layerdb/mounts`, pero no en `layerdb/sha256`. Docker separa de forma explícita las capas de imágenes de las capas de contenedores. El contenido de estos archivos nos da el *CacheID* correspondiente a esta capa, que podremos encontrar en `/var/lib/docker/overlay2` (o la carpeta que corresponda al driver que se esté usando). Con un simple comando *ls* veremos el contenido de la capa:

```
# ls
/var/lib/docker/overlay2/d5cb3c905cd0fc46c4552e3e38a967d74fd5d9357de1a
c719491b4da2e4e709a-init/diff/ -R
diff/:
dev  etc

diff/dev:
console  pts  shm

diff/dev/pts:

diff/dev/shm:

diff/etc:
hostname  hosts  mtab  resolv.conf
```

Como es lógico, al crear el contenedor, la capa `mount-id` estará vacía, pero los cambios que realicemos se reflejarán en:

```
/var/lib/docker/overlay2/
d5cb3c905cd0fc46c4552e3e38a967d74fd5d9357de1ac719491b4da2e4e709a/diff/
```

El contenido del archivo *parent* hace referencia al *ChainID* de la última capa de la imagen a partir de la cual se crea el contenedor, y podríamos obtener el DiffID del archivo:

```
/var/lib/docker/image/overlay2/layerdb/
sha256/5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10/
diff
```

Además de todas las capas de almacenamiento, se crean unos archivos de configuración para el contenedor:

```
/var/lib/docker/containers/
571a9e85e092fe77d52fa9fa014b94777ebbf7f4fc954d6a844768e958dfd505# tree
.
.
├── checkpoints
├── config.v2.json
└── hostconfig.json

1 directory, 2 files
```

La carpeta `checkpoints` se utiliza con el comando `docker checkpoint`, que es todavía experimental, y permitiría congelar un contenedor y migrar su estado a otro nodo.

Para ampliar la información sobre este comando, visítese:

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

Los otros dos archivos de configuración se generan a partir de una mezcla de la configuración de la imagen y los parámetros especificados a la hora de crear el contenedor.

¿Qué ocurre cuando ejecutamos docker run?

Al crear un contenedor a partir de una imagen Ubuntu con el siguiente comando:

```
$ docker run -it ubuntu /bin/bash
```

1. Docker descarga la imagen Ubuntu (`ubuntu:latest`) del registro por defecto (normalmente Docker Hub), a no ser que la imagen se encontrara ya localmente, de la misma manera que si hubiéramos hecho `docker pull ubuntu:latest`
2. Docker crea un nuevo contenedor (`docker container create`).

3. Docker crea una nueva capa sólo de lectura, con archivos especiales requeridos por el contenedor, y crea otra capa de lectura-escritura, la cual asocia como última capa del contenedor.
4. Se crea un interfaz de red que conecta el contenedor con la red por defecto, se asigna una dirección IP al contenedor y se establecen unas reglas de NAT por defecto, mediante iptables, para que el contenedor pueda conectarse a Internet.
5. Docker ejecuta el contenedor, y lanza el proceso `/bin/bash`. Como es un contenedor interactivo (opción `-i`) conecta la entrada estándar de nuestro terminal a la entrada estándar del proceso, y la opción `-t` hace que se cree un pseudo terminal conectado al contenedor (sin la opción `-t` únicamente se conectaría a la salida estándar).

> Se propone como ejercicio probar qué ocurre sin el `-t` y sin el `-i`. Comprobaremos que en un caso acepta la entrada pero no muestra prompt. En el otro, muestra prompt pero no acepta la entrada. Habrá que hacer `stop` y `rm` (remove).

Ejercicio: Crear nuestro propio contenedor

Como ejercicio avanzado (dejo los detalles al lector), podemos simular gran parte de lo que Docker hace para crear un nuevo contenedor.

```
# Vamos a usar primero docker para crear un contenedor Busybox, del
que extraeremos el sistema de archivos

$ CID=$(docker create busybox)

# La variable $CID contiene el ID del contendor

# Vamos a crear una carpeta temporal para extraer el sistema de
archivos de busybox

$ ROOTFS=$(mktemp -d)
$ echo $ROOTFS

# Docker export genera un .tar con el sistema de archivos del
contenedor, que extraemos en $ROOTFS

$ docker export $CID | tar -xf - -C $ROOTFS
$ ls $ROOTFS

# El resultado es que en la carpeta temporal localizada en $ROOTFS
tenemos un sistema de archivos completo del contenedor de Busybox

# Ahora empieza lo interesante. Todo lo anterior lo hemos hecho para
tener un sistema de archivos “base”, pero podríamos haberlo construido
de otras maneras.

# Aseguramos que el propietario de todos los archivos es “root”

$ sudo chown root:root $ROOTFS

# Generamos un UUID aleatorio

$ UUID=$(uuidgen)

# Con ese UUID aleatorio, creamos un cgroup de “cpu” y otro “memory”,
que limitarán los recursos de nuestro nuevo contenedor

$ sudo cgcreate -g cpu,memory:$UUID
```

```
# Establecemos el límite de memoria del cgroup

$ sudo cgset -r memory.limit_in_bytes=100000000 $UUID

# Establecemos los límites de CPU del cgroup

$ sudo cgset -r cpu.shares=512 $UUID
$ sudo cgset -r cpu.cfs_period_us=1000000 $UUID
$ sudo cgset -r cpu.cfs_quota_us=2000000 $UUID

# Y aquí viene la magia. Ejecutamos dentro de los cgroups que acabamos
de definir la instrucción “unshare”, que cambiará el proceso a un
nuevo “namespace”. Ya dentro del namespace establecemos el hostname,
montamos /proc, hacemos un chroot a $ROOTFS para que éste pase a ser
nuestro sistema de archivos raíz, y finalmente ejecutamos /bin/sh en
este nuevo sistema de archivos

$ sudo cgexec -g cpu,memory:$UUID \
     unshare --uts --ipc --net --pid --user --map-root-user --fork --
mount-proc \
     sh -c "/bin/hostname $UUID && mount -t proc proc $ROOTFS/proc &&
chroot $ROOTFS /bin/sh"

# Los siguientes comandos se ejecutan dentro del contenedor que hemos
creado

/ # echo "Hello from in a container"
Hello from in a container

/ # exit

# Al finalizar el proceso shell, que era el PID 1 dentro del namespace
del contenedor, se elimina el namespace y volvemos al proceso shell
orignal, fuera del contenedores

# Procedemos a limpiar el sistema de arcihvos raíz de la carpeta
temporal

$ rm -r $ROOTFS
```

Recursos adicionales para el ejercicio de crear nuestro propio contenedor:

1. Building a container from scratch in Go - Liz Rice (Microscaling Systems)

https://www.youtube.com/watch?v=Utf-A4rODH8

2. Cgroups, namespaces, and beyond: what are containers made from?

https://www.youtube.com/watch?v=sK5i-N34im8

A partir del minuto 41:11 y en adelante, se muestra un ejemplo más completo, incluyendo BTRFS y snapshot (copy-on-write)

CAPÍTULO 6

Optimización y buenas prácticas

Optimización y buenas prácticas

Algunos consejos generales de optimización y buenas prácticas ya se han ido comentando en apartados anteriores de esta guía. En este capítulo propondremos otros nuevos, de igual importancia y fundamentales para sacarle el máximo partido a Docker y aplicar las mejores prácticas.

Reducir el *build context*

Cuando hacemos "build" de una imagen (mediante el comando `docker build`), se envían por defecto al daemon Docker todos los ficheros de la carpeta especificada en el parámetro `context` (por ejemplo «`.`»):

```
$ docker build -t myimage -f Dockerfile .
Sending build context to Docker daemon   98.3mB
```

Si hay muchos ficheros en la carpeta, la transferencia de estos archivos hace más lento el proceso del *build*, y hace más fácil que nos equivoquemos al usar comandos `COPY` o `ADD`, añadiendo archivos innecesarios por error (como el propio Dockerfile). Es recomendable usar una carpeta separada para almacenar el contexto, e indicarlo explícitamente.

Si no necesitamos añadir ningún fichero al contenedor, podemos especificar «`-`» como carpeta del contexto para indicar un contexto vacío:

```
$ docker build -t myimage -f Dockerfile -
Sending build context to Docker daemon   2.56kB
```

Tamaño de la imagen

Debido al funcionamiento de los *union filesystems*, es importante recordar que al borrar un archivo en un paso del Dockerfile, no se elimina ese archivo de las capas anteriores de la imagen. El archivo sigue presente, pero no es accesible desde el contenedor. Por ejemplo:

- `L0: FROM ubuntu`
- `L1: RUN apt-get install` *`alguna-herramienta*`*
- `L2: RUN` *`algo-que-utiliza-la-herramienta para compilar o hacer algo`*
- `L3: RUN apt-get remove` *`alguna-herramienta`*

> Recordemos que union filesystem son sistemas de archivos en los que múltiples capas de una imagen, almacenadas en distintos directorios dentro del mismo host Linux, se presentan en un punto de montaje como un único directorio, combinando el contenido de todas las capas (Pag. 89)
>
> apt (Advanced Packaging Tool)-get : es un comando que permite instalar paquetes desde la línea de órdenes [52].

Las diferentes `L` son las capas (*layers*). Los paquetes que se instalan en el `apt-get install` y que crean la capa `L1` siguen existiendo aunque se eliminen en la capa `L3`. Por tanto, al descargar esta imagen y sus capas, estamos descargando todos esos archivos aunque sean inaccesibles.
Una solución sería intentar agrupar todos los `RUN` en una única capa, para que una vez finalizada la ejecución de ese paso, sólo nos genere una capa sin los archivos, que se habrán eliminado, quedando las capas de la siguiente manera:

- `L0: FROM ubuntu`
- `L1: RUN apt-get install` *`alguna-herramienta`* `&&` *`usar-la-herramienta`* `&& apt-get remove` *`alguna-herramienta`*

También veremos más adelante otra posibilidad que será usar lo que se denomina «*build multi-stage*.» (Capítulo página 157).

Seguridad

Al construir contenedores deberíamos aplicar las mismas prácticas de seguridad que para cualquier otro tipo de empaquetado y distribución de aplicaciones. Por ejemplo, nunca se deben crear contenedores con contraseñas (*passwords)* guardadas dentro del propio contenedor (no sólo en la última capa, sino en cualquier capa intermedia utilizada en la construcción).

Además uno de los problemas menos intuitivos que introduce la utilización de capas es que al borrar un archivo en una capa, no se borra el archivo de las capas precedentes. Este archivo todavía ocupa espacio, y puede ser accedido por cualquiera con las herramientas o conocimientos necesarios, simplemente accediendo a la carpeta del DiffID correspondiente.

Por tanto, datos secretos e imágenes no deberían mezclarse nunca. Credenciales y otros archivos confidenciales se deben proveer como parámetros, como variables de entorno con la opción "`-e`" o bien montando los volúmenes apropiados en el contenedor.

Veremos más consejos sobre seguridad en el Capítulo Seguridad en Docker de esta guía.

Optimización de la caché

Al ejecutar cada paso de un archivo Dockerfile, Docker examina cada instrucción, y mira si existe una imagen en su caché que pueda reutilizar en lugar de crear una nueva imagen duplicada. Es importante, por tanto, entender cómo se examinan las instrucciones y cuándo Docker puede, o no, encontrar una imagen reutilizable.

Partiendo de una imagen padre que ya está en la caché, se compara la instrucción con todas las imágenes derivadas de esa imagen padre para ver si alguna de ellas se construyó usando exactamente el mismo comando. Si no se encuentra, entonces **se invalida la caché, y por tanto esta capa y todas las demás deberán ser reconstruidas**.

Normalmente, con comparar el comando del Dockerfile es suficiente, pero algunas instrucciones requieren un examen más profundo:

- En las instrucciones `ADD` y `COPY` no se mira únicamente el nombre del archivo. Se calcula también un *checksum* (suma de verificación) sobre el contenido (no incluye las fechas de acceso y modificación del archivo). Si el contenido o los metadatos del archivo (permisos, etc.) han cambiado, la caché se invalida.
- Para el resto de comandos, sólo se mira el comando en sí, pero no el contenido de los ficheros generados. Por ejemplo, al procesar un `RUN apt-get …` no se examinan los ficheros actualizados o creados en el contenedor, solo se examina la cadena de texto utilizada.

Una vez que la caché se invalida para una capa, todos los siguientes comandos generan nuevas imágenes y la caché no se utiliza. Por tanto, habrá que volver a construir cada capa (lo que hará más lento el *build*), y estas nuevas capas tendrán que ser «*pusheadas*» al publicar y «*pulleadas*» al desplegar la imagen. Por este motivo es importante pensar cuidadosamente el orden de las capas y colocar los comandos o ficheros que vayan a sufrir muchos cambios en las

sucesivas versiones del contenedor al final del Dockerfile.

EJEMPLO:

Vamos a considerar dos imágenes:

Imagen A:

- `L0: FROM ubuntu`
- `L1: ADD source/* .`
- `L2: RUN apt-get install nodejs`
- `L3: ENTRYPOINT ["/usr/bin/node", "/main.js"]`

Imagen B:

- `L0: FROM ubuntu`
- `L1: RUN apt-get install nodejs`
- `L2: ADD source/* .`
- `L3: ENTRYPOINT ["/usr/bin/node", "/main.js"]`

Ambas imágenes se comportarán de forma idéntica, y de hecho, la primera vez que se haga un `pull` se descargarán el mismo número de capas con mismo tamaño en ambas. Sin embargo, ¿qué ocurre si cambiamos el archivo `main.js` de la aplicación?En el primer caso, las capas L1 y L2 serán invalidadas, y para un simple cambio en `main.js` tendremos que volver a descargar la capa que contiene la instalación de la aplicación `nodejs`. En el segundo caso, la capa L1 se cogerá de caché, y únicamente la capa L2 cambiará, por lo que el `push` y el `pull` de la imagen serán mucho más rápidos, y la utilización del espacio en disco será menor.

En general, para optimizar el *build* y el tamaño, interesa ordenar las capas, empezando por las más inmutables y terminando con las que más probablemente se actualizarán.

Para más información sobre optimización de la caché:

https://thenewstack.io/understanding-the-docker-cache-for-faster-builds/

Deshabilitar la caché

Hay casos en los que nos interesa forzar a que se invalide la caché. Por ejemplo, si no queremos usar paquetes cacheados en `apt-get` sino que queremos actualizar a la última versión.

En ese caso, podemos pasar la opción `--no-cache` al comando `docker build`, y todas las capas serán invalidadas y reconstruidas.

Cachés compartidas

La caché de Docker funciona únicamente de manera local. El motor de construcción de Docker genera imágenes intermedias para cada paso asociadas al comando ejecutado, y mediante esta asociación permite reutilizar capas ya existentes, pero **esas capas se guardan únicamente en `/var/lib/docker`, de manera local.**

Podemos utilizar el parámetro `--cache-from` para usar las capas de una imagen previamente construida como caché (pudiendo compartir la misma entre distintas máquinas). Esto será útil en casos en los que la caché de Docker vaya a dejar de funcionar:

- Si el entorno donde se ejecuta el *build* es efímero, y se elimina o limpia al finalizar la construcción, como puede pasar en algunas herramientas de integración continua.
- Si ejecutamos dos *builds* en hosts distintos, como construir la misma imagen por varios desarrolladores en distintas máquinas, o distintos agentes o ejecutores del sistema de integración continua.
- Si ejecutamos algún tipo de limpieza periódica, como `docker image prune` o `docker system prune` para eliminar imágenes no utilizadas.

Vamos a verlo con un ejercicio:

```
$ cat Dockerfile.base
FROM ubuntu
RUN apt-get update && apt-get install -y python

$ docker build -f Dockerfile.base -t airadier/cache-test:cache .
Sending build context to Docker daemon  4.096kB
Step 1/2 : FROM ubuntu
 ---> 549b9b86cb8d
Step 2/2 : RUN apt-get update && apt-get install -y python
 ---> Running in 43268496d49f
Get:1 http://archive.ubuntu.com/ubuntu bionic InRelease [242 kB]
…
Removing intermediate container 43268496d49f
 ---> d13132a3ecf5
Successfully built d13132a3ecf5
Successfully tagged airadier/cache-test:cache
$ docker push airadier/cache-test:cache

$ cat Dockerfile.app1
FROM ubuntu
RUN apt-get update && apt-get install -y python
RUN python -c "print('Hola app1')"

$ cat Dockerfile.app2
FROM ubuntu
RUN apt-get update && apt-get install -y python
RUN python -c "print('Hola app2')

$ docker build -t app1 -f Dockerfile.app1 .
Sending build context to Docker daemon  4.096kB
Step 1/3 : FROM ubuntu
 ---> 549b9b86cb8d
Step 2/3 : RUN apt-get update && apt-get install -y python
 ---> Using cache
 ---> d13132a3ecf5
Step 3/3 : RUN python -c "print('Hola app1')"
 ---> Running in 0cd9668d4256
Hola app1
Removing intermediate container 0cd9668d4256
 ---> fb2da59a742f
Successfully built fb2da59a742f
Successfully tagged app1:latest

$ docker image prune -a
…
```

```
$ docker build -t app1 -f Dockerfile.app2
<No funciona la cache>

$ docker image prune -a
…

$ docker pull airadier/cache-test:cache
$ docker pull airadier/cache-test:cache
cache: Pulling from airadier/cache-test
2746a4a261c9: Already exists
4c1d20cdee96: Already exists
0d3160e1d0de: Already exists
c8e37668deea: Already exists
bed948acc420: Pull complete
Digest:
sha256:5f206355c0a4e9d5e4c68e37c32cc9f682ec0ec700e7d89be9e3e992509d1eb
8
Status: Downloaded newer image for airadier/cache-test:cache
docker.io/airadier/cache-test:cache

$ docker build -t app2 --cache-from=airadier/cache-test:cache -f
Dockerfile.app2 .
Sending build context to Docker daemon  4.096kB
Step 1/3 : FROM ubuntu
 ---> 549b9b86cb8d
Step 2/3 : RUN apt-get update && apt-get install -y python
 ---> Using cache
 ---> d13132a3ecf5
Step 3/3 : RUN python -c "print('Hola app2')"
 ---> Running in 8acc59932ed6
Hola app2
Removing intermediate container 8acc59932ed6
 ---> ec0e53dc0987
Successfully built ec0e53dc0987
Successfully tagged app2:latest
```

¡Ojo! Estamos usando únicamente la caché especificada por `--cache-from`, y no la caché local. Si ejecutamos varias veces `build`, veremos que la última capa no se cachea.

Build multi-stage

Como ya hemos visto, uno de los problemas que nos podemos encontrar al construir un contenedor es que los archivos borrados siguen ocupando espacio.

Es muy cómodo y práctico instalar paquetes, ejecutar algunos comandos, y luego eliminarlos. O, por ejemplo, compilar nuestra aplicación como parte del propio proceso de construcción del contenedor, pero esto tiene el inconveniente de que nos deja mucho espacio ocupado por archivos borrados pero inaccesibles (o bien nos obliga a ejecutar múltiples comandos en un único `RUN` para no generar capas intermedias).

Para solucionar este problema, Docker introduce el concepto de un ***build multi-stage*** (multi etapa). Esto es, que dentro del mismo Dockerfile podemos generar múltiples imágenes, cada una de ellas denominada **etapa** (*stage*), y copiar contenidos de una etapa previa.

Por ejemplo:

```
FROM golang:1.12
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o main .
EXPOSE 8080
CMD ["./main"]
```

El contenedor construido está basado en el contenedor `golang:1.12`, que contiene las herramientas de desarrollo del lenguajeGo (golang). Copiamos primero los archivos `go.mod` y `go.sum` y ejecutamos el comando `go mod download` para descargar las dependencias, y así cachear estas capas. Luego copiamos el resto del código fuente, compilamos, y definimos el comando del contenedor.

El problema de esta imagen es que, sólo `golang:1.12`, ocupa unos 800MB. Además, tenemos en el contenedor todo el código fuente y los ficheros

intermedios de compilación. Aunque los borremos explícitamente, la imagen base seguirá ocupando de forma innecesaria.

Podemos solucionar esto convirtiéndolo en un *build multi-stage*:

```
FROM golang:1.12 as builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o main .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /app
COPY --from=builder /app/main .
EXPOSE 8080
CMD ["./main"]
```

La diferencia es que ahora, en la segunda imagen, partimos de la distribución Linux Alpine, que es mucho más ligera, y únicamente añadimos el archivo binario `/app/main` copiado de la etapa anterior. Así, la imagen generada en la etapa anterior será independiente, y su contenido no se incluirá en la imagen final.

> El archivo binario de código fuente y de aplicación es uno de los componentes que incluyen los contenedores, junto a la selección del sistema operativo y la información de configuración

Multi-stage y --cache-from

Anteriormente hemos visto que podemos utilizar el parámetro `--cache-from` para usar la caché desde una imagen, y poder compartirla entre distintas máquinas. El problema de este método es que las capas de las etapas intermedias generadas en *build multi-stage* no se incluyen en la imagen final, y por tanto no es posible reusarlas.

Para ello, existe una alternativa que consistiría en usar el flag «`--target`» del comando `docker build` para construir una etapa intermedia, en lugar de la etapa final. En este caso podríamos ejecutar:

```
$ docker build --target=builder -t airadier/cache-test:builder .
```

Esto nos generaría y etiquetaría una imagen (con el resultado de construir únicamente el target «`builder`») con las capas correspondientes a esa imagen.

Podríamos enviar esa imagen a un registro:

```
$ docker push airadier/cache-test:builder
```

Y descargarla en otra máquina para construir la imagen reusando la caché de la imagen *builder*:

```
$ docker pull airadier/cache-test:builder

$ docker build -t my-go-app --cache-from airadier/cache-test:builder .
```

Véase el ejercicio completo en el siguiente enlace: «Caching Docker layers on serverless build hosts with multi-stage builds, --target, and –cache-from»

https://andrewlock.net/caching-docker-layers-on-serverless-build-hosts-with-multi-stage-builds---target,-and---cache-from/

Cachés distribuidos

Alternativamente, otros sistemas de construcción como BuildKit (accesible en las últimas versiones de Docker con el CLI plugin *buildx*) o Kaniko, utilizan otros mecanismos para compartir cachés entre varios hosts (cachés distribuidos).

Véase más información sobre este mecanismo en BuildKit y en Kaniko en los siguientes enlaces:

- BuildKit/buildx: «Docker build cache sharing on multi-hosts with BuildKit and buildx»

  https://medium.com/titansoft-engineering/docker-build-cache-sharing-on-multi-hosts-with-buildkit-and-buildx-eb8f7005918e

- Kaniko:

  https://cloud.google.com/cloud-build/docs/kaniko-cache

Contenedores «*Distroless*»

Una de las principales ventajas de los contenedores, su ligereza y menor tamaño que una máquina virtual, se difumina si construimos nuestros contenedores sobre distribuciones Linux completas (como Ubuntu, Debian o Alpine, aunque algunas, como Alpine, sean muy ligeras).

El proyecto ***Distroless Docker Images*** de Google proporciona un conjunto de imágenes mínimas, de las cuales se han eliminado gestores de paquetes, *shells* (intérpretes de comandos) y otros programas típicos que se encuentran en distribuciones de Linux, con el objetivo de incluir contenedores que contengan únicamente la aplicación principal y sus dependencias de ejecución. Disponen de distintas variantes de imágenes *Distroless* para distintos tipos de aplicaciones:

- *static-debian10:* para binarios 100% estáticos, sin dependencia de la biblioteca libc.
- *cc-debian10:* para binarios casi estáticos, con dependencia únicamente de la biblioteca glibc,
- *base-debian10:* binarios que dependen únicamente de los software glibc, libssl y openssl.
- *java-debian10:* contiene un runtime OpenJDK mínimo.
- Y en estado experimental:
  - Python2.7
  - Nodejs
  - Dotnet
  - Python3
  - Jetty

Si generamos binarios 100% estáticos, como con el lenguaje C o usando ciertas opciones al compilar Go, Rust, etc., es incluso posible tener contenedores sin ningún tipo de base, únicamente incluyendo el binario de la aplicación a ejecutar

Para más información acerca de Distroless, consultar:

https://github.com/GoogleContainerTools/distroless

One-Concern Container y desacoplamiento

El concepto de «*one-concern container*» es una recomendación de buenas prácticas que indica que cada contenedor debería ocuparse únicamente de un solo asunto. Esto no quiere decir que solo deba ejecutar un proceso, ya que por ejemplo un contenedor con el servidor *apache* ejecutará múltiples procesos para gestionar las peticiones que reciba. Sin embargo, una aplicación web completa podría estar compuesta de tres contenedores separados, cada uno con su propia imagen, para gestionar, por un lado, el servidor web, por otro, la persistencia en una base de datos, y por otro, una memoria caché (como el motor de base de datos Redis), todo de manera desacoplada.

Es posible lanzar múltiples procesos dentro de un contenedor usando herramientas como el gestor de procesos *Supervisord*[12] o con un «*script wrapper*». Pero este desacoplamiento entre aplicaciones hace más fácil escalar horizontalmente y reutilizar contenedores.

> Un *wrapper script* es un script que contiene un montón de otros scripts y comandos en el. [53]

Cuando se ejecutan múltiples aplicaciones en un contenedor, éstas pueden tener distintos ciclos de vida, o encontrarse en distintos estados. Por ejemplo, el contenedor puede estar *running* (creado), pero uno de sus componentes principales puede haber muerto o no responder. Sin definir un «healthcheck» adicional, el gestor de contenedores (ya sea Docker o un orquestador como Kubernetes) no pueden saber si el contenedor está ejecutándose correctamente, y reiniciarlo si fuera necesario.

12 http://supervisord.org

Contenedores efímeros

Los archivos Dockerfile, como ya se ha insistido anteriormente, deberían generar contenedores lo más efímeros posibles, en el sentido de que debería ser posible detener y destruir un contenedor, reconstruirlo y reemplazarlo por una nueva versión prácticamente sin ninguna preparación o configuración. Esto es posible al tener también un desacoplamiento entre los datos usados o generados por el contenedor, y la aplicación en sí.

Logging

Suele ser habitual que el proceso principal (y único, exceptuando procesos hijos, *workers*, etc.) que se ejecuta en el contenedor, genere sus mensajes de ***log*** por la salida estándar en lugar de escribir en archivos dentro del contenedor. Si los logs se escribieran «a disco», es decir, en archivos dentro del contenedor, éstos se perderían cuando el contenedor se destruyera, salvo si las carpetas donde se escriben los logs se crearan sobre volúmenes persistentes.

Por lo tanto, es mucho más práctico que los registros salgan por la salida estándar, y dejar que el sistema de log de Docker, configurable, los guarde de forma persistente.

Existen múltiples drivers de logs en Docker que se pueden especificar al arrancar el daemon, por ejemplo:

- Json-file
- Journald
- Awslogs
- Otras
- Syslog
- Fluentd
- Splunk

En la versión *community* de Docker, no todos los drivers permiten además usar *docker log*, únicamente los drivers: *local, json-file* y *journald*.

Para más información a cerca del logging, consúltese:

https://docs.docker.com/config/containers/logging/configure/

Testing

Dentro del proyecto de GoogleContainerTools, podemos encontrar la herramienta `container-structure-test`, que nos permite automatizar las pruebas sobre la estructura de un contenedor., por ejemplo dentro de nuestro Pipeline de CI/CD. CI/CD es un método de integración/distribución continua para implementar, integrar y distribuir aplicaciones a los clientes mediante la automatización en las etapas del desarrollo de aplicaciones [CI/CD, RedHat].

> En un proceso Pipeline, mientras una instrucción es ejecutada, otra está siendo interpretada por el ordenador y una más está siendo leída. Pipeline, es un esquema que interpreta un flujo constante de trabajo de forma secuencial, dando como entrada de cada proceso la salida del anterior de forma concatenada. [54]
>
> La automatización de la infraestructura, consiste en el uso de sistemas de software para crear instrucciones y procesos repetibles a fin de reemplazar o reducir la interacción humana con los sistemas de TI.[55]

Podemos definir en un fichero con formato YAML (.yml) o JSON (.json) una serie de pruebas a ejecutar sobre un contenedor, como ejecutar comandos y comparar la salida de éste para verificar su correcto funcionamiento, o validar versiones, verificar la existencia de archivos, metadatos del contenedor, etc.

Para más información, consultar:

https://github.com/GoogleContainerTools/container-structure-test

Entrypoint estándar

CMD vs ENTRYPOINT

Vamos a aclarar la diferencia entre las instrucciones `CMD` y `ENTRYPOINT` a la hora de construir un contenedor.

> Recordatorio:
>
> - `CMD`: Esta instrucción provee de valores por defecto al contenedor. Cabe destacar que sólo tendrá validez la última aparición del comando.
> - `ENTRYPOINT`: Esta instrucción permite pasar cualquier argumento al punto de entrada, pero anulará cualquier elemento especificado con CMD

Tanto la instrucción `CMD`, como `ENTRYPOINT` (así como `RUN`, aunque en el caso de esta instrucción su propósito es completamente distinto) aceptan dos formatos:

- **Formato shell** → `CMD "comando"`
- **Formato exec** → `CMD ["ejecutable", "par1", "par2", …]`

En el formato **shell**, Docker realmente ejecuta:

```
/bin/sh -c <comando>
```

Por lo que ocurren cosas como reemplazos de variables:

Si definimos en formato **shell**:

```
CMD "echo Hola $NOMBRE"
```

Salida: "`Hola <valor de NOMBRE>`"

Mientras que si definimos en formato **exec**:

```
CMD ["/bin/echo", "Hola", "$NOMBRE"]
```

Salida: "`Hola $NOMBRE`".

La instrucción **CMD** permite definir un **comando por defecto para el contenedor**, que se ejecutará en caso de que el usuario no especifique nada al ejecutar el contenedor. Si definimos:

```
CMD “echo Hola mundo”
```

y lanzamos el contenedor con:

```
$ docker run -it miimagen
```

Salida “Hola mundo”.

Sin embargo, si ejecutamos:

```
$ docker run -it miimagen /bin/bash
```

El comando definido por CMD se ignorará completamente, y se ejecuta /bin/bash

ENTRYPOINT sin embargo **convierte un contenedor en ejecutable,** y permite definir un punto de entrada al que el usuario puede añadir parámetros. Si definimos:

```
ENTRYPOINT [“/bin/echo”, “Hola”]

$ docker run -it miimagen
```

Salida “Hola”.

Pero si ejecutamos:

```
$ docker run -it miimagen /bin/bash
```

Salida “Hola /bin/bash”,
ya que los parámetros recibidos se añaden como parámetros adicionales.

Aquí, `ENTRYPOINT` se combina con `CMD` de forma que si ambos están definidos, `CMD` indica los **parámetros por defecto** que recibe el entrypoint:

```
ENTRYPOINT [“/bin/echo”, “Hola”]
CMD [“Mundo”]


$ docker run -it miimagen
```

Salida: “`Hola Mundo`”

```
ENTRYPOINT [“/bin/echo”, “Hola”]
CMD [“Mundo”]

$ docker run -it miimagen /bin/bash
```

Salida: “`Hola /bin/bash`”

Si definimos `ENTRYPOINT` en formato **shell**, los argumentos son ignorados, tanto los proporcionados por `CMD` como los que el usuario pase al ejecutar el contenedor:

```
ENTRYPOINT “echo Hola”
CMD [“Mundo”]

$ docker run -it miimagen
```

Salida: “`Hola`”

```
$ docker run -it miimagen /bin/bash
```

Salida: “`Hola`”

Parámetro por defecto y comandos alternativos

Aunque no está documentado ni es un requisito imprescindible, es habitual encontrar en muchas imágenes un *script bash* (script que ha de ejecutarse

usando la shell BASH), o similar, con una serie de comandos que se ejecuta como *entrypoint*, y que permite al usuario pasar opciones al proceso principal o lanzar un proceso completamente distinto (por ejemplo un shell).

Podemos ver un ejemplo en el siguiente enlace:

https://github.com/jenkinsci/docker/blob/master/jenkins.sh

En este caso, el script verifica si hay argumentos en la ejecución del script. En caso de que no haya, o de que el primero de ellos vaya prefijado por «--», lanza Jenkins, pasándole los argumentos adicionales que el usuario ha especificado.

Sin embargo, si el primer argumento no empieza por «--», ejecuta este proceso:

```
…

# if `docker run` first argument start with `--` the user is passing
jenkins launcher arguments
if [[ $# -lt 1 ]] || [[ "$1" == "--"* ]]; then

  # read JAVA_OPTS and JENKINS_OPTS into arrays to avoid need for eval
(and associated vulnerabilities)
  java_opts_array=()
  while IFS= read -r -d '' item; do
    java_opts_array+=( "$item" )
  done < <([[ $JAVA_OPTS ]] && xargs printf '%s\0' <<<"$JAVA_OPTS")

 …

  exec java -Duser.home="$JENKINS_HOME" "${java_opts_array[@]}" -jar $
{JENKINS_WAR} "${jenkins_opts_array[@]}" "$@"
fi

# As argument is not jenkins, assume user want to run his own process,
for example a `bash` shell to explore this image
exec "$@"
```

Gestión de señales y procesos zombie

Para entender los problemas de la gestión de señales y de los procesos zombie, y cómo podemos actuar ante ellos, debemos primero de comprender lo siguiente:

Por la forma en la que funcionan los contenedores, cada uno aislado en un namespace de procesos, el primer proceso que se ejecuta en el contenedor recibe el PID 1 (process id 1). Este es el único proceso con el que Docker o Kubernetes se pueden comunicar enviando señales (`SIGTERM`, `SIGKILL`, `SIGINT`, etc.). Estas señales son importantes para indicar al proceso que debe detenerse, y pueda así salir limpiamente de el. En un sistema operativo normal, el PID 1 sería el proceso `init`, que está preparado para gestionar estas señales de forma correcta.

Por otra parte, dentro de un contenedor, puede ocurrir que:

- El PID 1 sea el shell, si se ha ejecutado el proceso mediante shell form.
- El PID 1 sea el ejecutable de una aplicación, si se ha ejecutado el proceso mediante exec form.

Esto puede generar problemas como la mala gestión de señales o los procesos zombies, los cuales se comentan a continuación.

Gestión de señales

Si al enviar las señales `SIGTERM` o `SIGINT` al proceso con el PID 1, este dispone de un gestor para estas señales, el proceso responderá a las mismas ejecutando la rutina asignada a ellas. Pero si, por el contrario, el proceso con el PID 1 no ha registrado gestores para estas señales, como ocurre cuando el PID 1 es un *shell*, estas señales serán ignoradas, y por tanto nuestra aplicación no podrá gestionar la señal para hacer un *graceful* exit, es decir, salir de forma ordenada y controlada. Docker enviará directamente la señal `SIGKILL` pasado el periodo de gracia, matando así al PID 1 y a todo el contenedor.

> Periodo de gracia (grace period), es el tiempo que dura la cuenta regresiva iniciada en el momento en el que se envía la señal SIGTERM al proceso principal. Este es el tiempo que tiene un contenedor para apagar correctamente la aplicación en ejecución y salir, antes de que se envíe la señal SIGKILL y el contenedor termine ``violentamente´´. [56]

Por lo tanto, es recomendable como buena práctica usar la forma **exec** para que el PID 1 sea nuestra aplicación, dependiendo así de ella, que será la que se encargará de gestionar estas señales.

Además, si la forma **exec** ejecuta un script de inicialización antes de lanzar la aplicación, es importante usar el comando `exec` para reemplazar el proceso en lugar de crear un proceso hijo:

```
#!/usr/bin/bash
python /app/server.app
```

Frente a:

```
#!/usr/bin/bash
exec python /app/server.app
```

En el primer caso, el proceso `bash` será el PID 1 y `python` será un proceso hijo. En el segundo caso, `python` reemplaza al proceso `bash`, convirtiéndose en el PID 1, y recibiendo, por tanto, las señales directamente.

Procesos zombies

El segundo problema es el de los llamados **procesos *huérfanos* o procesos *zombies*,** esto es, un proceso hijo cuyo proceso padre ha muerto.

¿Cómo puede ocurrir esto? Si nuestra aplicación está bien realizada, esta debería hacer un *wait* (llamada del sistema para esperar la finalización de un proceso) sobre los procesos hijos finalizados antes de terminar, sin generar así procesos huérfanos. Pero pueden ser dependencias externas o bibliotecas de terceros las que generen estos procesos. Un ejemplo típico puede ser un agente de *Jenkins*, que ejecuta multitud de procesos shell, herramientas de desarrollo, etc. que pueden llegar a generar procesos huérfanos.

Cuando un proceso queda huérfano, es heredado por el proceso PID 1. En un sistema operativo normal, el proceso `init` que es el PID 1 tiene un mecanismo para eliminar los procesos huérfanos. Pero si nuestra aplicación no está preparada para heredar y gestionar los procesos Zombies, estos se irán acumulando, consumiendo algunos recursos del sistema de forma indeseada.

Para solucionar esto podemos hacer que nuestra aplicación gestione estos procesos, o podemos hacer que el PID 1 no sea nuestra aplicación, sino un gestor de procesos como «Tini», que se encarga de solucionar ambos problemas, tanto gestionar las señales y pasarlas a la aplicación como eliminar los procesos zombies.

Desde Docker 1.13, se puede especificar el parámetro "`--init`" al comando `docker run`, que fuerza a que el PID 1 de nuestro contenedor no sea el comando o el *entrypoint*, sino el primer ejecutable `docker-init` que se encuentra en el PATH (ruta) del sistema, y que por defecto es una implementación de Tini.

Para más información sobre Tini, consultar:

https://github.com/krallin/tini

Límite de memoria contenedor vs Kernel OOM

Tanto Docker como Linux tienen un mecanismo llamado el OOM (Out of Memory) Killer, encargados de matar procesos cuando el sistema se está quedando sin memoria.

Si **no especificamos límite de memoria** a un contenedor, el uso de ésta puede crecer de forma indefinida, pudiendo entrar en funcionamiento el OOM Killer del kernel. Este, a través de un algoritmo, determina qué proceso/s eliminar para liberar memoria. Cualquier proceso que no sea crítico puede ser candidato para ser eliminado, incluyendo el daemon Docker, *sshd*, o muchos otros. Cuando esto sucede la situación es crítica y el sistema puede quedar inutilizable.

Pese a que Docker ajusta algunos parámetros del proceso y de los contenedores para que el algoritmo de selección tenga más probabilidad de elegir como víctima un contenedor en lugar del proceso dockerd, es **muy recomendable establecer límites de memoria** en casos en los que el uso no esté perfectamente acotado.

Para ello, debemos usar opciones como **`--memory [-m]`**, para indicar la cantidad máxima de memoria RAM que puede usar el contenedor o `--memory-swarp`, para establecer la cantidad de memoria que este contenedor puede intercambiar en el disco duro.

Al establecer los parámetros `--memory` y `--memory-swap`, hay que tener en cuenta que estos tienen diferentes efectos cuando se usan solos que cuando se configuran en conjunto:

- Si se define `--memory` y también `--memory-swap`, la segunda opción es la memoria **total** y `--memory` la cantidad de memoria sin swap.

  Por ejemplo: `--memory 1g --memory-swap 2g`

  Significa que el contenedor puede usar 2GB en total, siendo 1GB de swap. Usando `--memory 1g --memory-swap 1g` se deshabilita el uso

de la swap (para un total de 1GB).

- Si se define `--memory` pero **no** `--memory-swap`, éste último parámetro toma el mismo valor.

  Por ejemplo: `--memory 1g`

  Permite usar 1GB de física y 1GB de Swap (2GB en total).

- Estableciendo `--memory-swap -1` se da acceso ilimitado a memoria swap.

Véase el siguiente enlace para ampliar la información anterior:

https://docs.docker.com/config/containers/resource_constraints/#limit-a-containers-access-to-memory

El comando **`docker stats`** nos permite ver en tiempo real el uso de memoria vs el límite de memoria de los contenedores, a falta de herramientas de monitorización más potentes como Prometheus + CAdvisor.

Si la memoria física en un host es limitada, y la suma de los límites de memoria física de los contenedores supera la memoria física del host, pero hay swap suficiente, entonces se envía a swap también parte de la memoria de los contenedores.

A continuación de muestran algunas recomendaciones para evitar el OOM Killer del kernel*:*

- Disponer en el **host** de tanta memoria física como el footprint del sistema sin contenedores ejecutándose, más la suma del uso habitual de los contenedores.

  > El footprint, o huella de memoria, es la cantidad de memoria que requiere un programa cuando se ejecuta [57]

- Añadir límites totales y razonables a los contenedores (`--memory` y `--memory-swap`) para contemplar picos de consumo, pero matar un contenedor si hace un uso excesivo.
- Añadir suficiente memoria swap al sistema, a modo de buffer de

emergencia. Esta debe ser igual a la suma del footprint base, más la suma de los totales de memoria de los contenedores, menos la memoria física existente. Con esta regla nos aseguraremos de que, aunque podamos sufrir ralentización debido al *swapping* (transferencia de memoria), el OOM Killer del kernel nunca llegará a eliminar procesos críticos:

SWAP = ∑(memoria contenedor n) + footprint sistema – ram fisica

El buffer es el espacio de memoria en el que se almacenan datos de forma temporal, en espera de ser transferidos entre dos ubicaciones. Se utiliza para evitar que el programa o recurso que los necesita se quede sin datos durante una transferencia. [58]

CAPÍTULO 7

Docker Compose

Docker Compose

Tras conocer los conceptos y funcionalidades básicas de Docker, y haber hecho un repaso de las recomendaciones de su uso y sus buenas prácticas, vamos a ver a continuación el funcionamiento de la **herramienta Docker Compose**.

Docker Compose nos permite definir y correr aplicaciones que necesitan utilizar diferentes contenedores, automatizando la creación de la misma con una única ejecución. Así, nos ofrece la posibilidad de documentar y configurar todas las dependencias de la aplicación con la sencillez de una única ejecución para poder levantar un entorno completo.

Esta herramienta es muy útil en desarrollo, pruebas y flujos de trabajo de integración continua (aquella que requiere la confirmación de código de forma periódica en un repositorio compartido) .

En Docker Compose el fichero por defecto debería de nombrarse como «docker-compose.yml», aunque tenemos la posibilidad de nombrarlo de cualquier forma que deseemos.

Además, se puede usar un fichero docker-compose.yml de tal manera que éste recurra a un Dockerfile (véase Capítulo de esta guía), lo cual le permitiría crear implementaciones de contenedores más complejas, usando estos dos ficheros en conjunto. [59]

Más información sobre Docker Compose:

https://docs.docker.com/compose/

https://docs.docker.com/compose/compose-file/

Sintaxis del fichero de Docker Compose

La sintaxis del fichero para la creación de contenedores se realiza en formato yaml (.yml) y tiene una serie de campos obligatorios, así como una serie de opciones que nos pueden resultar de utilidad en determinados casos. A continuación vemos un ejemplo de este fichero, para posteriormente explicar cada una de las opciones de configuración disponibles para la creación del mismo.

Ejemplo de fichero Docker Compose:

```
version: "3.7"
services:
  <resource-name>:
    context: ./dir
      dockerfile: Dockerfile-alternate
      args:
        buildno: 1
    labels:
“name=value”
    image: <image-name>:<version>
    ports:
      - "<source-port>:<dest-port>"
    environment:
      <var-name>: <var-value>
    volumes:
      - "<source-path>:<container-path>"
    links:
      - <resource-name-to-link>
    command: ["command", "arg1", "argn"]
    networks:
      -  name
    cap_add:
      - ALL
    cap_drop:
-  NET_ADMIN
```

En los siguientes epígrafes pasaremos a estudiar cada una de las opciones de configuración para la creación del Docker Compose que este nos ofrece, tomando como referencia el ejemplo anterior.

Versión [version]

Con la etiqueta «**version**» se identifica la versión de Docker Compose que se desea utilizar en nuestro fichero. A continuación se muestra una tabla con las versiones disponibles, siendo siempre recomendable utilizar la última de ellas, actualmente la 3.7 [año 2020], debiendo escribirse de la siguiente manera: `version: "3.7"`

Tabla 4: Versiones de Docker Compose

Compose file format	Docker Engine release	Compose file format	Docker Engine release
2.2	1.13.0+	3.3	17.06.0+
2.3	17.06.0+	3.4	17.09.0+
2.4	17.12.0+	3.5	17.12.0+
3.0	1.13.0+	3.6	18.02.0+
3.1	1.13.1+	**3.7**	**18.06.0+**
3.2	17.04.0+		

Creación de imagen [build]

A través de esta opción se pueden indicar las diferentes posibilidades de configuración de la imagen a la hora de su construcción. Es posible indicar el directorio sobre el cual deseamos que se recoja el fichero Dockerfile (véase capítulo , en la página 34 de esta guía) para la creación de la imagen, simplemente indicando la ruta del mismo:

```
version: "3.7"
services:
  webapp:
    build: ./dir
```

O también es posible indicar objetos debajo del mismo para identificar un fichero Dockerfile con un nombre diferente. En este caso, para indicar la ruta de este, se utilizaría el parámetro **`context`**:

```
version: "3.7"
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
```

Además, también es posible indicar el nombre y el *tag* que se desea asignar a la imagen creada por este método, simplemente indicando el nombre `<image-name>` y el tag `<image-tag>` de la misma, tal y como vemos a continuación:

```
build: ./dir
image: <image-name>:<image-tag>
```

Imagen [image]

La opción «**`image`**» indicará la imagen y versión que se desea utilizar para la generación del contenedor, cuya sintaxis sería la siguiente:

```
    image: <image-name>:<version>
```

En caso de no indicar una versión, se utilizará el valor por defecto, siendo este «`latest`», es decir, la última versión disponible de la misma.

Argumentos [args]

Existe la posibilidad de añadir **argumentos** de creación de nuestras imágenes que sólo sean accesibles durante este proceso de creación.

Para ello, en primer lugar, será necesario que estos argumentos se indiquen previamente en el fichero Dockerfile, utilizando el comando **`ARG`** de la siguiente manera:

```
ARG buildno
ARG gitcommithash

RUN echo "Build number: $buildno"
RUN echo "Based on commit: $gitcommithash"
```

Y, posteriormente, en Docker Compose, así:

```
build:
  context: .
  args:
    buildno: 1
    gitcommithash: cdc3b19
```

Etiquetas [labels]

Podemos añadir metadatos (**etiquetas**) a nuestras imágenes y contenedores con la idea de poder identificarlas de una forma más sencilla. La sintaxis en este caso sería la siguiente:

```
labels:
    - "com.example.description=Accounting webapp"
    - "com.example.department=Finance"
- "com.example.label-with-empty-value"
```

Etapa de construcción de destino [target]

«`target`» permite la construcción de la etapa (stage) de una imagen especificada tal y como se define dentro del Dockerfile.

```
build:
  context: .
  target: prod
```

Añadir o quitar capacidades [cap_add , cap_drop]

Se pueden añadir o eliminar privilegios o capacidades concretas de los contenedores, otorgando a los usuarios y grupos de usuarios poder sobre ellos. Para ello, la sintaxis en el Compose sería la siguiente:

```
cap_add:
  - ALL

cap_drop:
  - NET_ADMIN
  - SYS_ADMIN
```

[Obsérvese el listado de las capacidades que se pueden añadir o eliminar en el punto la Tabla 8: Privilegios que pueden ser añadidos o eliminados de contenedores. de esta guía.]

Control groups [cgroups]

Es posible indicar el padre del *cgroup* [véase capítulo Control Groups (Cgroups)] que se desea utilizar en el contenedor con la opción `cgroup_parent`, tal y como vemos en el siguiente ejemplo:

```
cgroup_parent: <parent-name>
```

Sobreescribir comando [command]

Es posible sobreescribir el comando de la imagen utilizada para levantar el contenedor configurado por defecto en el fichero Dockerfile. La sintaxis sería similar a la que se utilizaría en dicho fichero:

```
command: <command-to-execute>
```

También sería posible indicar el mismo en forma de lista:

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

Sobrescribir Entrypoint [entrypoint]

Otra de las posibilidades, al igual que en `command`, sería sobreescribir el entrypoint [véase punto Entrypoint estándar] especificado por defecto en el Dockerfile:

```
entrypoint: /code/entrypoint.sh
```

Como en la opción de configuración `command` que hemos visto en el punto anterior, también sería posible indicar los comandos en forma de lista:

```
entrypoint:
    - php
    - -d
    - zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20100525/xdebug.so
    - -d
    - memory_limit=-1
- vendor/bin/phpunit
```

Configuracion específica [config]

Se pueden añadir también **configuraciones específicas** para los diferentes servicios levantados mediante Docker Compose, existiendo dos formas de hacerlo, una corta y otra larga:

La **forma corta** ofrece la posibilidad de incluir un fichero de configuración que posteriormente sea cargado en dicho servicio:

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
```

```
    configs:
      - my_config
      - my_other_config
configs:
  my_config:
    file: ./my_config.txt
  my_other_config:
    external: true
```

La **forma larga**, permitiría añadir también dicha configuración en el propio servicio, además de poder cargar un fichero de configuración, cuyas opciones se pueden editar o modificar.

Esta forma de aplicar la configuración permite añadir los siguientes parámetros:

- `source` → el nombre de la configuración existente en Docker
- `target` → el nombre del fichero de configuración que se montará en `/run/configs`. Si no se especifica tomará el valor de `source`
- `uid` → el id del usuario
- `gid` → el id del grupo
- `mode` → los permisos aplicados a dicho fichero. Por defecto, en la versión 1.13.1, el valor de esta directiva es de «0000». En versiones superiores, será «0444».

Así quedaría con la forma larga

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    configs:
      - source: my_config
        target: /redis_config
        uid: '103'
        gid: '103'
        mode: 0440
configs:
  my_config:
```

```
    file: ./my_config.txt
  my_other_config:
    external: true
```

Credenciales [secrets]

La opcion **secrets** opción permitiría añadir un fichero con las credenciales utilizadas para el servicio.

> Un «secreto» es una pieza de información requerida para tareas de autenticación, autorización, cifrado, etc. Es información que debe ser confidencial y es indispensable. Por ejemplo, para que usuarios y sistemas puedan interconectarse de manera segura.[60]

Al igual que en la opción de `configs`, permite añadir dicha configuración mediante una forma corta y mediante una larga.

La **versión corta** permite indicar un fichero que será posteriormente cargado en el servicio:

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - my_secret
      - my_other_secret
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

La **opción larga** permitiría, además, añadir cierta configuración específica para el mismo.

Esta forma de aplicar la configuración permite añadir los siguientes parámetros:

- `source` → el nombre de la configuración existente en Docker

- `target` → el nombre del fichero de secretos que se montará en `/run/secrets`. Si no se especifica tomará el valor de source
- `uid` → el id del usuario
- `gid` → el id del grupo
- `mode` → los permisos aplicados a dicho fichero. Por defecto, en la versión 1.13.1, el valor de esta directiva es de «0000». En versiones superiores, será «0444».

```
version: "3.7"
services:
  redis:
    image: redis:latest
    deploy:
      replicas: 1
    secrets:
      - source: my_secret
        target: redis_secret
        uid: '103'
        gid: '103'
        mode: 0440
secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

Contenedores [Container_name]

Por defecto, Docker Compose identifica los diferentes contenedores con el nombre «`resources_resourcename`». Esto puede modificarse con la directiva **`container_name`**, tal y como se muestra en el siguiente ejemplo:

```
container_name: my-container-name
```

Dependencias

Cabe la posibilidad de añadir dependencias a los diferentes servicios

desplegados. Es decir, se puede indicar que para que un contenedor se cree, sea necesario que previamente se hayan creado otros contenedores que este necesita para su correcto funcionamiento.

- `docker-compose up` → arranca los servicios en el orden de dependencias establecido.
- `docker-compose up SERVICE-NAME` → arrancaría el servicio indicado y, previamente al mismo, levantaría los contenedores de los cuales es dependiente.
- `docker-compose stop` → para los contenedores en el orden de dependencia establecido, sin eliminarlos.

Podemos ver a continuación un ejemplo de cómo se aplicaría esta opción:

```
version: "3.7"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

Recursos [resources]

La opción de «`resources`» permitiría especificar la configuración de CPU y Memoria que se desea aplicar a un contenedor a la hora de su creación. Además de especificar un límite en el uso de la misma, es posible especificar una reserva en caso de ser necesaria.

En el siguiente ejemplo, el servicio de almacenamiento en memoria `redis` está obligado a usar no más de 50M[13] de memoria y 0.50 (50% de un solo núcleo) de

13 M es la unidad de medida de memoria conocida como «Longitud de *palabra*». Indica el número invariable de bits que posee cada *palabra,* entendiéndose esta como la cadena finita de bits que son manejados como un conjunto o unidad por una máquina

tiempo de procesamiento disponible (CPU), y tiene 20M de memoria y 0.25 de tiempo de CPU reservado (como siempre disponible).

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

Dispositivos [devices]

Se puede indicar un listado de los dispositivos mapeados en el servicio (como podría ser, por ejemplo, una memoria USB):

```
devices:
  - "/dev/ttyUSB0:/dev/ttyUSB0"
```

Domain Name System [dns]

Se pueden indicar **DNSs** propios a la hora de crear el servicio, permitiendo la modificación de los mismos si fuese necesario.

```
dns:
  - 8.8.8.8
  - 9.9.9.9
```

[61]

Redes [network]

Se puede especificar el modo de configuración red que se desea utilizar a la hora de crear un contenedor, permitiendo incluso referenciar una red ya creada previamente con un nombre determinado. Esto ofrece una posibilidad más de personalización del servicio deseado:

- `network_mode: "bridge"`
- `network_mode: "host"`
- `network_mode: "none"`
- `network_mode: "service:[service name]"`
- `network_mode: "container:[container name/id]"`

```
services:
  some-service:
    networks:
     - some-network
- other-network
```

Puertos [ports]

Se pueden especificar los puertos utilizados por el daemon para la comunicación con el contenedor, así como también indicar la iP a través de la cual se comunican, o el protocolo utilizado para tal fin.

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "127.0.0.1:8001:8001"
 - "6060:6060/udp"
```

Volúmenes [volumes]

Se pueden indicar también volúmenes utilizados en la máquina host para que sean montados en una ruta específica del contenedor, permitiendo con ello poder

extraer datos del mismo o incluir los mismos en el contenedor. Además, es posible indicar el tipo de volumen que se desea montar, así como con qué permisos debería de montarse el mismo.

```
db:
    image: postgres:latest
    volumes:
     - "/var/run/postgres/postgres.sock:/var/run/postgres/
postgres.sock:ro"
     - "dbdata:/var/lib/postgresql/data"
```

Para consultar más información sobre volúmenes:

https://docs.docker.com/compose/compose-file/#volumes

Variables de entorno [environment]

Cabe la posibilidad de añadir variables de entorno que posteriormente sean utilizadas en nuestros contenedores. Para ello se deberá seguir la siguiente sintaxis:

```
environment:
  VAR1: development
  VAR2: 'true'
  VAR3: 12
```

Enlaces [links]

Es posible enlazar los diferentes contenedores entre sí, e incluso asignar un alias a los mismos con el fin de identificarlos de una forma más sencilla. La sintaxis en este caso sería «`SERVICE:ALIAS`», tal y como se aprecia en el siguiente ejemplo:

```
web:
  links:
   - db
   - db:database
   - redis
```

Logs

Para la configuración de **logs** se pueden utilizar diferentes servicios, como puede ser `syslog`, `awslogs`, etc. Esto nos permite dar un punto más de personalización a los mismos e incluso configurar un agente de dicho servicio a la hora de generar el contenedor, indicando también en este caso la dirección donde se encuentra el servidor utilizado para el almacenamiento de los mismos. Se puede observar un listado de los servicios existentes en el punto .

A continuación un ejemplo de cómo debería de configurarse:

```
logging:
  driver: syslog
  options:
    syslog-address: "tcp://192.168.0.42:123"
```

Chequeo [healthcheck]

Es posible especificar chequeos periódicos a los contenedores con la idea de obtener una indicación del estado en el que se encuentran, y que posteriormente estos datos puedan ser enviados a una herramienta encargada de su monitorización.

La sintaxis en este caso sería la siguiente:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost"]
  interval: 1m30s
  timeout: 10s
  retries: 3
  start_period: 40s
```

En caso de haber habilitado el `healthcheck` desde el fichero Dockerfile, también es posible deshabilitarlo de la siguiente forma:

```
healthcheck:
```

```
  disable: true
```

Límites de uso [ulimits]

La utilidad `ulimits` *(usage limits)* permite establecer límites al uso de ciertos recursos del sistema. Con Docker Compose se pueden establecer estos límites sobreescribiendo los valores por defecto aplicados por `ulimits` a los contenedores. Se debería especificar en este caso los límites `soft/hard` de los mismos como un entero para una asignación.

```
ulimits:
  nproc: 65535
  nofile:
    soft: 20000
    hard: 40000
```

Comandos útiles con Docker Compose

Existen **otros comandos que se deben usar en Docker Compose** con otros propósitos una vez completada la creación de nuestro fichero:

Tabla 5: Comandos Docker Compose

Comando	Utilidad
`docker-compose up -d`	Para construir el entorno:
`docker-compose up -f filename -d`	O, en el caso de haber llamado al fichero con un nombre diferente a «`docker-compose.yml`», añadir la opción `-f`
`docker-compose ps`	Para ver los contenedores que se encuentran en ejecución (una vez completada la creación de los mismos):
`docker-compose ps -a`	Para ver todos los contenedores disponibles estén o no arrancados (añadiendo la opción -a):
`docker-compose up` `docker-compose down`	Para parar o levantar los contenedores
`docker-compose down -v`	Para parar y eliminar los contenedores
`docker-compose rm <container-`	Para eliminar un contenedor parado

Comando	Utilidad
`name|id>`	
`docker-compose exec <container-name|id> /bin/bash`	Para acceder a los contenedores (una vez completados los pasos anteriores)
`docker-compose logs -ft`	Para ver los logs de los contenedores (de todos ellos al mismo tiempo)
`docker-compose logs -ft <container-name|id>`	Para ver los logs de un contenedor específico (indicando el nombre o identificador de este)

Práctica Docker Compose

De forma análoga a la práctica realizada en el apartado sobre el Dockerfile, se propone en este apartado como práctica la creación de 3 contenedores diferentes con los servidores Nginx, Tomcat y el gestor de bases de datos MySQL, esta vez utilizando Docker Compose.

Los 3 contenedores indicados quedarán de la siguiente manera:

Contenedor Tomcat:

```
  # tomcat node
  tomcat:
    image: tomcat:8.5.50-jdk8
    ports:
      - "8080:8080"
    environment:
      JDBC_DDBB: example_db
      JDBC_URL: jdbc:mysql://10.100.1.200:3306/example_db?
connectTimeout=0&amp;socketTimeout=0&amp;autoReconnect=true
      JDBC_USER: db_user
      JDBC_PASS: db_password
    volumes:
      - "./tomcat/code/webapps:/usr/local/tomcat/webapps"
      - "./tomcat/logs:/usr/local/tomcat/logs"
    links:
      - db
```

Contenedor Ngix:

```
  # nginx node
  nginx:
    image: nginx:1.17
    ports:
      - "80:80"
    volumes:
      - "/var/run/docker.sock:/tmp/docker.sock:ro"
      - "./nginx/conf:/etc/nginx/conf.d"
      - "./nginx/html:/usr/share/nginx/html"
      - "./nginx/logs:/var/log/nginx"
    links:
      - tomcat
```

Contenedor Mysql:

```
version: "3.7"
services:
  # mysql database
  db:
    image: mysql:5.6
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: mysqlpassword
      MYSQL_DATABASE: example_db
      MYSQL_USER: db_user
      MYSQL_PASSWORD: db_password
    volumes:
      - "./mysql/data:/docker-entrypoint-initdb.d"
      - "./mysql/logs:/var/log/mysql"
```

CAPÍTULO 8

Construir imágenes en contenedores

Construir imágenes en contenedores

Hay ocasiones en las que puede ser necesario construir o incluso ejecutar contenedores dentro de un propio contenedor. Por ejemplo, un *pipeline* de integración continua en el que las etapas se ejecutan dentro de un contenedor, y a la vez queremos crear la imagen que vamos a desplegar, o compilar dentro de un entorno que ejecutamos en un contenedor.

Vamos a analizar dos mecanismos que nos permiten resolver ambos casos, aunque no son muy recomendables por cuestiones de seguridad: en caso de que necesitemos construir imágenes dentro de contenedores, deberíamos usar herramientas alternativas como Kaniko o BuildKit. Y si tenemos la necesidad de ejecutar un contenedor dentro de otro contenedor, tal vez debamos replantearnos el flujo de trabajo y, por ejemplo, convertir nuestro entorno de desarrollo en una etapa más del *pipeline*, en vez de ejecutarlo como un contenedor dentro de otra etapa más genérica.

Docker-in-docker

El servicio Docker-in-docker (DinD) permite que el motor Docker funcione como un contenedor dentro de Docker, requieriendo la instalación completa de Docker en su interior.

Lo primero que tendremos que hacer, tal y como se muestra a continuación, será ejecutar Dind en modo privilegiado, ya que los archivos del sistema serán montados desde el sistema host. [62]:

```
$ docker run --privileged --name df-docker -d docker:17.06.0-dind

$ docker run --rm -it --link df-docker:docker docker:18.02.0 sh

# docker version
```

El entrypoint del segundo contenedor establece la variable `DOCKER_HOST=tcp://docker:2375` y eso permite, mediante el enlazado de contenedores, conectar al daemon que se ejecuta en el primer contenedor.

Para más información, véase el siguiente enlace:

https://github.com/docker-library/docker/blob/master/docker-entrypoint.sh

Aunque Docker-in-docker (DinD) funciona, tiene varios inconvenientes:

- Dependiendo de los módulos de seguridad (AppArmor, SELinux, etc.) el contenedor interior puede tener conflictos con los perfiles aplicados a contenedores exteriores, haciendo que no funcione.
- El contenedor exterior se ejecuta sobre un filesystem estándar, pero el interior se ejecuta sobre un *coppy-on-write filesystem*, lo que puede causar problemas con algunas combinaciones (AUFS no funciona sobre AUFS, etc.)
- No se puede reutilizar la caché del *host*. Intentar montar

`/var/lib/docker` en el contenedor DinD puede llevar a corrupción de datos, ya que el daemon está pensado para acceso exclusivo a estos archivos.

Más información:

- «Docker in Docker»
  https://hub.docker.com/_/docker/
- «Using Docker-in-Docker for your CI or testing environment? Think twice»
  https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/

Docker-out-of-docker

Como alternativa a Docker in Docker (DinD) podemos montar el socket de Docker dentro del contenedor:

```
$ docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock
docker:18.02 sh

# docker version
```

Ventajas:

- No se requiere ejecutar ningún contenedor en modo privilegiado.
- Se puede reutilizar la caché del host
- Es fácil de configurar

Inconvenientes:

- Aunque no requiere modo privilegiado, no es más seguro, porque estamos ejecutando Docker sobre el host, con lo que ¡podemos conseguir acceso completo al host (como hemos visto anteriormente)!
- No permite gestionar un entorno completamente limpio de Docker, al compartir el daemon. Si queremos un entorno limpio, necesitaríamos DinD.

Alternativas

Si únicamente tenemos la necesidad de construir imágenes dentro de un contenedor, y no de ejecutar, existen varias herramientas alternativas a DinD y Docker-out-of-docker que permiten construir una imagen que sigue la especificación OCI, sin depender de un daemon Docker. Entre ellas, y de manera muy resumida:

BuildKit (by Docker):

https://github.com/moby/buildkit

- Permite ejecutar partes del *build* en paralelo para los multi-stage
- Funcionalidades planeadas: *build* distribuido, con caché compartida
- Funcionalidad experimental: rootless mode, que permitiría ejecutar el Docker daemon y los contenedores como un usuario no root, en aras de mitigar posibles vulnerabilidades en el daemon y en el container runtime
- Se integra a partir de Docker 18.06 con `DOCKER_BUILDKIT=1`
- Es rápido, obsérvese en el siguiente gráfico comparativo de la Figura 26.

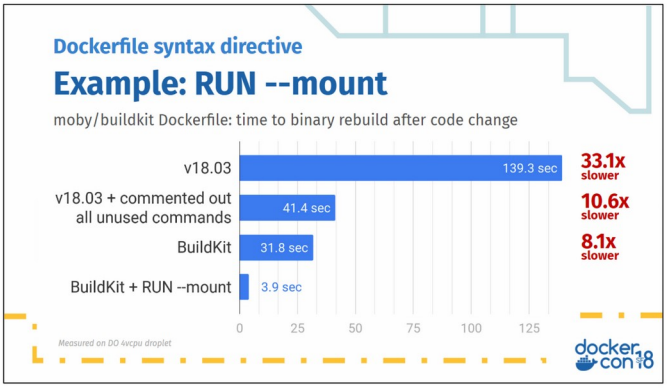

Figura 26: Diferencia de velocidad entre BuildKit y construcción tradicional de Docker

img:

https://github.com/genuinetools/img

- Usa BuildKit como biblioteca, pero sin daemon (*daemonless*), y con CLI (línea de comandos)
- Rootless

Buildah (by Redhat):

https://buildah.io/

- Soporta Dockerfile
- *Daemonless*
- Rootless

Kaniko (by Google):

https://github.com/GoogleContainerTools/kaniko

- Se ejecuta en un contenedor, pero no privilegiado
- Ejecuta instrucciones RUN dentro de su rootfs y namespace
- No se recomienda si puede haber Dockerfiles maliciosos, por falta de aislamiento

Y otras herramientas, como **Bazel**, **Orca**, **Source-to-Image, Metaparticle**.

CAPÍTULO 9

Registros de Docker

Registros de Docker

Docker Hub y alternativas

Docker Hub es el registro por defecto utilizado por Docker en el caso de que no se especifique un prefijo. Ofrece alojamiento gratuito para imágenes públicas, sin límite de repositorios o tags, y otras funcionalidades como los *builds* automáticos, que construyen una imagen y hacen `push` al repositorio a partir del código fuente.

Además de Docker Hub, existen otros registros alternativos. Vamos a examinar a continuación ventajas y desventajas de algunos de ellos, empezando por el propio Docker Hub.

Docker Hub

- Registro por defecto.
- Alojamiento ilimitado para repositorios públicos, de modo gratuito.
- Permite crear un webhook (evento que desencadena una acción [¿Que es un webhook?[14]] para cada repositorio, de forma que se lance un POST a una URL cuando se hace un *push* al repositorio.
- En el plan gratuito se incluye un repositorio privado y un *build* paralelo.
- Permite organizar repositorios por organizaciones. Una organización permite agrupar varios usuarios con distintos permisos (*pull/push/create,* etc.)
- Planes de pago desde 7$/mes por 5 repositorios privados y 5 *builds* paralelos.

14 https://es.mailjet.com/blog/news/que-es-webhook/

- No tiene versión On-Prem (instalación en servidor dentro de la propia empresa), aunque el registro *vanilla* [véase punto] puede usarse, es código abierto.

Para saber más:

https://hub.docker.com

Quay

- Adquirido por Redhat.
- Dispone de versión Cloud y On-Prem.
- Gratis para repositorios públicos. Desde 15$/mes por 5 repositorios privados.
- *Builds* automáticos y escaneo de imágenes de contenedores para detectar vulnerabilidades conocidas.
- Soporta múltiples mecanismos de notificación además de un webhook (como email o la aplicación Slack) y distintos eventos (*push*, *build*, vulnerabilidad encontrada, …)
- Es un registro de contenedores Docker, y de “aplicaciones” Helm (herramienta de gestión de aplicaciones).

Para saber más:

https://guay.io

Amazon Elastic Container Registry (ECR)

- No tiene un UI (interfaz de usuario) para encontrar imágenes tipo Docker Hub o Quay. Está más orientado a un uso privado.
- Registros privados por defecto, hay que aplicar IAM (administrador de

identidades y de acceso a servicios y recursos) para conceder permisos de acceso

- Cada repositorio tiene su URL, (larga y complicada de recordar).
- Precio por almacenamiento y transferencia: consúltese:

https://aws.amazon.com/ecr/pricing/

Para saber más:

https://aws.amazon.com/es/ecr/

Google Container Registry

- Sus URLs son del tipo `gcr.io/user/repositorio`
- Tampoco dispone de UI para encontrar contenedores, pero dispone de URLs públicas y privadas
- Precio también por almacenamiento y transferencia.

Para saber más:

https://grc.io

Azure Container Registry

- Pago por día, incluye unos mínimos de almacenamiento y transferencia. Luego, pago por uso adicional.
- Permite almacenar contenedores, Helm Charts (gráficos para la gestión de aplicaciones complejas), y otros artefactos OCI.
- Integración con *pipelines, builds, image scanning, content trust* (utilización de firmas digitales para el envío y recepción de datos), …

Para saber más:

https://azure.microsoft.com/es-es/services/container-registry/

Harbor

Harbor es un proyecto OpenSource, basado en el registro oficial de Docker y miembro de la CNCF (Cloud Native Computing Foundation), originalmente desarrollado por la compañía VMware. Se basa en una serie de componentes que, unidos, consiguen proporcionar:

- Un registro de imágenes de contenedores (basado en Docker Registry) y de Charts Helm (basado en el proyecto *Chart Museum*)
- RBAC (Role Based Access Control) método para restringir el acceso del sistema a los usuarios autorizados.
- Replicación entre distintos Harbor, con posibilidad de definir políticas de replicación (filtros, *pull/push*, por eventos, etc.)
- Multi-tenant o multi-inquilino: arquitectura que permite a una sola instancia de software servir a muchos clientes [63]
- Escaneo de vulnerabilidades en imágenes (con el proyecto Clair)
- Soporte de autentificación digital mediante los protocolos LDAP o OpenID Connect
- Borrado de imágenes y recolector de basura (aunque el Docker Registry también lo proporciona mediante la API y mediante un comando especial)
- Content Trust, autenticación de imágenes (firmado) - proyecto Notary[15]
- Interfaz gráfico para gestión, con portal de usuario
- Auditoría de operaciones
- API REST (además de la propia API del registro) para operaciones administrativas.

15 https://github.com/theupdateframework/notary

El código del proyecto se aloja en: https://github.com/goharbor/harbor

Para saber más: https://goharbor.io/

JFrog Artifactory

- Versión Open-Source y versión Pro
- Almacén de "*artifacts*" de todo tipo (.jar, maven, npm, python, nuget, etc.) que también soporta imágenes Docker en la versión Pro (en la versión Opern Source, sólo Maven, Gradle e Ivy

> Artifact: tipo de subproducto producido durante el desarrollo del software. Ayudan a describir la función, la arquitectura y el diseño del software, o están relacionados con el proceso de desarrollo en sí mismo.[64]

Para saber más:

https://jfrog.com/open-source/).

Sonatype Nexus

1. Similar a Artifactory, también almacena de binarios y *artifacts*.
2. Versión Open Source: Maven/Java, npm, NuGet, RubyGems, Docker, P2, OBR, APT y YUM, …
3. La versión Pro ofrece características como High Availability (continuidad operacional), métodos adicionales de autenticación, o plugins.

Para saber más:

https://www.sonatype.com/product-nexus-repository

Ejercicio: instalación de Harbor

A modo de práctica, se propone la instalación del registro Harbor, siguiendo los siguientes pasos:

- ✔ Descargamos el instalador online de la siguiente dirección, y lo extraemos en una carpeta, donde procederemos a ejecutar los siguientes pasos.:

https://github.com/goharbor/harbor/releases/download/v1.10.0/harbor-online-installer-v1.10.0.tgz

- ✔ Instalamos Docker Compose

https://docs.docker.com/compose/install/

```
sudo curl -L \
 "https://github.com/docker/compose/releases/download/1.25.3/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

- ✔ Editamos el archivo harbor.yml y ajustamos algunos parámetros:
  - hostname: bastion
  - http.port: 8080
  - Comentamos el bloque “https” del YAML. No deberíamos hacer esto en un entorno de producción, pero nos simplificará la realización del ejercicio práctico.

```
$ ./install.sh
```

- ✔ El script ejecuta un contenedor, y crea `docker-compose.yml` y la carpeta `common/`. En caso de problemas con permisos, habrá que ejecutarlo como **root** o revisar el yaml del punto anterior para comprobar que las rutas configuradas pueden ser utilizadas por el usuario.

✔ Examinar **docker-compose.yml** para ver qué ha generado el script de instalación. Algunos ejemplos

- Opción `:z` de los mounts → SELinux label (content is shared among multiple containers)
- Opción `dns_search?` → Como hemos puesto "bastion" sin un dominio completo, la opción, se queda simplemente como «`.`»

✔ Iniciamos Harbor con docker-compose a partir del archivo generado:

```
✔ $ sudo docker-compose up -d
✔ ...
✔ Creating harbor-log ... done
✔ Creating harbor-portal ... done
✔ Creating registry      ... done
✔ Creating harbor-db     ... done
✔ Creating registryctl   ... done
✔ Creating redis         ... done
✔ Creating harbor-core   ... done
✔ Creating harbor-jobservice ... done
    Creating nginx          ... done
```

✔ Abrimos el navegador accediendo a la IP de la máquina:

- http://10.100.1.200:8080/
- Usuario: admin
- Password: Harbor12345 (si no lo hemos cambiado en harbor.yml)

✔ Navegamos por la interfaz, creamos un proyecto **curso-docker**, un usuario, etc.

✔ Echamos en falta el scanner, para habilitarlo, y que aparezca en el docker-compose, hay que ejecutar:

```
$ ./prepare --with-clair
```

✔ Hacemos un `docker-compose down` y `up`, y esta vez tenemos Clair disponible.

- ✔ ¿Dónde se almacenan los datos? Ver los volúmenes en el `docker-compose.yml` - se está almacenando en `/data` (en el raíz del sistema de archivos del host, si no lo hemos cambiado).

```
$ docker tag alpine bastion:8080/curso-docker/alpine
$ docker push bastion:8080/curso-docker/alpine
```

- ✔ Nos requerirá hacer un *docker login* para poder hacer el `push`
- ✔ Revisar ahora la imagen en la UI.

Alojar nuestro propio registro

Puede tener sentido alojar nuestro propio registro On-Prem (o en la nube pero únicamente accesible desde nuestra propia infraestructura) en casos en los que necesitemos mayor confidencialidad de las imágenes que alojamos. El proceso puede ser más o menos sencillo dependiendo del software utilizado y los requisitos de nuestro alojamiento. Debemos plantearnos algunas preguntas:

- ¿Necesitamos que la comunicación sea segura y confidencial, y por tanto tenemos que usar TLS/SSL?

  En caso afirmativo, deberemos generar y configurar unos certificados TLS. Docker por defecto exige que todos los registros sean seguros, a menos que lo deshabilitemos implícitamente como veremos en el siguiente punto.

- ¿Necesitamos un UI para administración, búsqueda de contenedores, etc.?

  Si es así, tendremos que valorar opciones comerciales tipo Nexus o Artifactory, o bien Harbor, si no necesitamos otro tipo de artefactos y requerimos alta disponibilidad. En otros casos, el registro Docker básico puede ser suficiente.

- ¿Es requisito tener autorización basada en grupos, roles, o similar?

  El registro Docker básico solo ofrece autentificación mediante *basic authentication*. Para casos más avanzados es necesario poner un proxy delante, gestionando la autenticación y autorización. Harbor y otras soluciones ofrecen estas características de serie.

Registros inseguros

En caso de alojar nuestro propio registro, lo más recomendable es habilitar TLS y utilizar un certificado firmado por una CA (autoridad certificadora) de confianza, ya sea de pago o gratuita tipo Let's Encrypt.

Si el registro no utiliza TLS o usamos un certificado auto-firmado, deberemos configurar Docker para aceptar estos casos, como veremos a continuación.

Más información:

https://docs.docker.com/registry/insecure/

Registro sin TLS (HTTP plano)

Para usar un registro sin TLS deberemos añadir este registro como *inseguro* de forma explícita en el fichero de configuración del daemon Docker. En Linux, este fichero de configuración se encuentra por defecto en `/etc/docker/daemon.json` y deberemos añadir una entrada con la dirección y puerto del registro a utilizar:

```
{
  "insecure-registries" : ["myregistrydomain.com:5000"]
}
```

La entrada `insecure-registries` contiene una lista de cadenas con el nombre del dominio y puerto, separados por comas, en formato de lista (array) JSON.

En Windows, el fichero se encuentra en En la aplicación Docker Desktop se puede configurar desde las preferencias de la aplicación, en el menú correspondiente. Será necesario reiniciar el daemon Docker para aplicar los cambios.

Registro con certificado auto firmado

Esta opción es más segura que un registro sin TLS, ya que utiliza cifrado en la comunicación. Debido a que utiliza un certificado auto firmado, debemos añadir éste como autoridad certificadora de confianza.

Así, generamos el certificado, por ejemplo, con OpenSSL:

```
openssl req \
  -newkey rsa:4096 -nodes -sha256 -keyout domain.key \
  -x509 -days 365 -out domain.crt
```

Y arrancamos el registro usando la clave y certificado correspondiente.

Si intentamos hacer `pull`/`push` (descargar/subir) desde el registro, Docker fallará al no reconocer el certificado como firmado por una CA de confianza.

En Linux, debemos copiar el archivo `domain.crt` a:

```
/etc/docker/certs.d/myregistrydomain.com:5000/ca.crt
```

En Windows y otros sistemas, debemos añadir el certificado al almacén de autoridades certificadoras raíz de confianza. Para ello, pueden seguirse las instrucciones detalladas en este enlace[16]:

En algunas distribuciones Linux puede ser necesario también añadir el certificado a las CAs de confianza del sistema, al utilizar estas distribuciones bibliotecas como OpenSSL que verifican los certificados con las CAs de este almacén.

En la distribución Ubuntu:

```
$ cp certs/domain.crt
/usr/local/share/ca-certificates/myregistrydomain.com.crt
$ update-ca-certificates
```

16 https://www.realsystems.com.mx/blog/servicio-en-la-nube-15/post/administrar-certificados-raiz-de-confianza-en-windows-10-297

Docker Registry "vanilla"

https://docs.docker.com/registry/deploying/

Utilizando los siguientes comandos podemos ver cómo ejecutar un contenedor de la imagen *registry:2*, que es la última versión del registro básico de Docker.

```
$ docker run -d -p 5000:5000 --restart=always --name registry
registry:2
```

Una vez el contenedor está en ejecuciónpodemos intentar *pushear* imágenes a este registro. En el ejemplo, tageamos la propia imagen *registry:2* como *localhost:5000/registry:2* y ejecutamos docker push:

```
$ docker tag registry:2 localhost:5000/registry:2

$ docker push localhost:5000/registry:2
```

Vamos a ver qué ocurre si intentamos pushear usando la dirección IP del contenedor (en el ejemplo es 10.0.2.15) en lugar de “localhost”:

```
$ docker tag registry:2 10.0.2.15:5000/registry:2

$ docker push 10.0.2.15:5000/registry:2
The push refers to repository [10.0.2.15:5000/registry]
Get https://10.0.2.15:5000/v2/: http: server gave HTTP response to
HTTPS client
```

Ejercicio: ¿Cómo solucionarlo? → Pista: ver Insecure registries

Ejercicio: Echar un vistazo a la carpeta `/var/lib/registry/` dentro del contenedor del registro (`docker exec -ti registry sh`).

¿Qué ocurre si destruimos el contenedor? ¡Al destruirlo y volverlo a arrancar, todos los datos de /var/lib/registry se han perdido! Un contenedor no tiene persistencia.

¿Cómo evitarlo? → Pista: volúmenes en contenedores

Ejercicio: hacer el almacenamiento persistente

```
$ docker run -d \
  -p 5000:5000 \
  --restart=always \
  --name registry \
  -v /mnt/registry:/var/lib/registry \
  registry:2

..
```

Drivers de almacenamiento

https://docs.docker.com/registry/storage-drivers/

El registro de Docker permite configurar distintas opciones de almacenamiento para las imágenes almacenadas. Por defecto se almacenan en el sistema de archivos del contenedor, en /var/lib/registry, pero soporta otras opciones ,como almacenar en memoria, en un *bucket S3*, etc:

- Filesystem (por defecto)
- Inmemory
- S3 / Azure / GCS

- Swift / OSS
- Otros.

Se configuran con opciones de arranque o en el `config.yml`

https://docs.docker.com/registry/configuration/

Autenticación básica

En los apartados previos, hemos lanzado el registro sin ningún tipo de autentificación, por lo que cualquier usuario anónimo podría *pushear* y *pullear* imágenes.

```
$ docker run -d -p 5000:5000 --restart=always -e
REGISTRY_AUTH_HTPASSWD_REALM=myrealm -e
REGISTRY_AUTH_HTPASSWD_PATH=/etc/docker/htpasswd --name registry
registry:2
6766d72be1be20a71753517719834e87928b19f01f085ea3d36014e6391faa7d
```

A continuación, vemos cómo ejecutar el registro habilitando un tipo de autentificación básico, con usuario y contraseña. Al no especificar un usuario y contraseña, se generará uno aleatoriamente, que podemos ver en el log del contenedor:

```
vagrant@bastion:~$ docker logs registry
time="2020-02-03T23:43:32.088469331Z" level=warning msg="No HTTP
secret provided - generated random secret. This may cause problems
with uploads if multiple registries are behind a load-balancer. To
provide a shared secret, fill in http.secret in the configuration file
or set the REGISTRY_HTTP_SECRET environment variable."
go.version=go1.11.2 instance.id=fe2fcf18-3270-4193-96ef-5726e7edc082
service=registry version=v2.7.1
time="2020-02-03T23:43:32.088557505Z" level=info msg="redis not
configured" go.version=go1.11.2 instance.id=fe2fcf18-3270-4193-96ef-
5726e7edc082 service=registry version=v2.7.1
time="2020-02-03T23:43:32.088649471Z" level=info msg="Starting upload
purge in 7m0s" go.version=go1.11.2 instance.id=fe2fcf18-3270-4193-
96ef-5726e7edc082 service=registry version=v2.7.1
time="2020-02-03T23:43:32.094915414Z" level=info msg="using inmemory
blob descriptor cache" go.version=go1.11.2 instance.id=fe2fcf18-3270-
4193-96ef-5726e7edc082 service=registry version=v2.7.1
```

```
time="2020-02-03T23:43:32.148195801Z" level=warning msg="htpasswd is
missing, provisioning with default user" go.version=go1.11.2
password=laDAPktdJBMxO6grd5S1Ame3bC3X172u60mA5JkhQPE user=docker
time="2020-02-03T23:43:32.148311138Z" level=info msg="listening on
[::]:5000" go.version=go1.11.2 instance.id=fe2fcf18-3270-4193-96ef-
5726e7edc082 service=registry version=v2.7.1
```

Como indicábamos, y el propio registro del contenedor nos indica, al lanzarlo sin montar el fichero `/etc/docker/httpasswd` ni la opción `REGISTRY_HTTP_SECRET`, se crea un usuario Docker con un password por defecto.

```
$ docker login 10.0.2.15:5000
Username: docker
Password:
```

Ejercicio: montar fichero `htpasswd` y dar de alta unos usuarios

Ejercicio: crear certificado y arrancar Docker con TLS habilitado. Probar a quitar de `insecure-registries`.

Métricas prometheus

El registro soporta **prometheus**, especificando una opción para lanzar el contenedor del registro con métricas de **prometheus** habilitadas:

```
$ docker run -d -p 5000:5000 -p 5001:5001 --restart=always \
    -e REGISTRY_HTTP_DEBUG_PROMETHEUS_ENABLED=true \
    -e REGISTRY_HTTP_DEBUG_ADDR=0.0.0.0:5001 \
     --name registry registry:2
```

Almacenamiento interno

Se propone como ejercicio muy interesante para el lector el examinar la carpeta donde el contenedor almacena las imágenes (/var/lib/registry por defecto).

Ejercicio: pushear algunas imágenes sencillas al registro, observar los digests de las imágenes subidas, y tratar de entender, analizando las carpetas de blobs y de manifests, la relación entre el digest del manifest, el archivo de configuación, las capas de la imagen. Etc.

(in)Mutabilidad de tags

Que un *tag* sea **inmutable,** significa que en el momento en el que hemos hecho `pull` de una imagen con ese tag, no es posible subir otra imagen del mismo repositorio y tag. Por tanto, el tag no puede nunca mutar y apuntar a una imagen distinta. Hemos visto en el UI de Harbor que existe una opción para hacer ciertos tags inmutables, pero este no es el comportamiento por defecto.

> Recordemos que un tag en Docker es una etiqueta que se utiliza para identificar las versiones de las imágenes, siendo este tag el alias de su ID.[12]

El caso más típico de tag *mutable* es el uso del tag **`latest`** para apuntar a la última versión disponible. Pero por ejemplo podemos usar tags como `repositorio:1.10` para apuntar a la última minor versión (última versión desarrollada) disponible (1.10.0, luego 1.10.1, 1.10.2, etc). O por ejemplo para etiquetar las imágenes usadas en distintos entornos (dev, qa, staging, prod, …).

El hecho de que los tags sean mutables nos aporta estas posibilidades, pero también nos genera una serie de problemas. No tenemos garantía de que dos procesos usando el mismo `repositorio:tag` en un momento dado estén utilizando la misma imagen, lo cual es contraintuitivo.

Para evitar esto, Docker nos permite aprovechar el hecho de que los propios manifests de las imágenes son contenido indexado mediante el `sha256`. Por tanto, igual que hacemos:

```
$ docker pull alpine:latest
```

también podremos hacer:

```
$ docker pull
alpine@sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b
11c4367d
```

Para referirnos a la misma imagen, podemos verlo con la salida de `docker inspect`:

```
$ docker inspect alpine:latest
[
    {
        "Id":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
6a",
        "RepoTags": [
            "bastion:8080/curso-docker/alpine:latest",
            "alpine:latest"
        ],
        "RepoDigests": [

"bastion:8080/curso-docker/alpine@sha256:ddba4d27a7ffc3f86dd6c2f92041a
f252a1f23a8e742c90e6e1297bfa1bc0c45",

"alpine@sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172
b11c4367d"
        ],
        "Parent": "",
…
```

Ese `sha256` es el *digest* del **manifest** de la imagen y, como también la hemos subido a nuestro registro local, podemos encontrar el manifest de la imagen:

```
bastion:8080/curso-docker/
alpine@sha256:ddba4d27a7ffc3f86dd6c2f92041af252a1f23a8e742c90e6e1297bf
a1bc0c45
```

En:

`/data/registry/docker/registry/v2/blobs/sha256/dd/ddba4d27a7ffc3f86dd6c2f92041af252a1f23a8e742c90e6e1297bfa1bc0c45/data`

```
{
   "schemaVersion": 2,
   "mediaType":
"application/vnd.docker.distribution.manifest.v2+json",
   "config": {
      "mediaType": "application/vnd.docker.container.image.v1+json",
      "size": 1511,
      "digest":
"sha256:e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a9977
```

```
6a"
   },
   "layers": [
      {
         "mediaType":
"application/vnd.docker.image.rootfs.diff.tar.gzip",
         "size": 2802957,
         "digest":
"sha256:c9b1b535fdd91a9855fb7f82348177e5f019329a58c53c47272962dd60f71f
c9"
      }
   ]
}
```

Esto permite que, por ejemplo Kubernetes, a la hora de procesar la creación de contenedores a partir de su definición, traduzca el formato `repositorio:tag` a `repositorio@sha256:<digest>`. De esta forma, aunque la imagen cambie y el contenedor sea reconstruido o migrado a otros hosts, se seguirá **utilizando siempre la misma imagen**.

Pero esto tiene a su vez otro inconveniente. El registro, para soportar el acceso a manifests por *digest*, y la posibilidad de que los tags muten, debe guardar los manifests de las imágenes viejas, así como todas las capas referenciadas.

Ejercicio: *pushear* dos variantes de una misma `imagen:tag`, con una modificación, y analizar las siguientes carpetas:

```
/data/registry/docker/registry/v2/repositories/curso-docker/alpine/
_manifests
/data/registry/docker/registry/v2/repositories/curso-docker/alpine/
_manifests/revisions
/data/registry/docker/registry/v2/repositories/curso-docker/alpine/
_manifests/tags
/data/registry/docker/registry/v2/repositories/curso-docker/alpine/
_manifests/tags/<tag>/current
/data/registry/docker/registry/v2/repositories/curso-docker/alpine/
_manifests/tags/<tag>/index
```

La misma imagen tiene dos *digest* distintos (el manifest) en dos repositorios distintos debido a que el propio nombre repositorio:tag forma parte del cálculo del

digest del manifest, por lo que al hacer `push` (subir) a un registro distinto, el *digest* del manifest cambia.

Más información:

https://docs.docker.com/registry/spec/api/#manifest

Garbage Collection

La recolección de basura, o ***garbage collection*** (GC) es una forma de administración automática de memoria, mediante la cual el recolector intenta recuperar la basura o la memoria ocupada por objetos que el programa ya no usa.

Funcionamiento

El Garbage Collection del registro de Docker oficial se puede lanzar mediante un comando, y funciona de la siguiente manera:

- Cada capa (blob) es referenciada por uno o varios manifests, que a su vez son referenciados por tags.
- Cuando se borra un Manifest, usando la API[17] para borrar la imagen por *digest,* ese manifest deja de referenciar a una serie de capas, aunque al ser compartidas, algunas de esas capas pueden estar todavía referenciadas por otros manifests.
- El proceso de recolección de basura se ejecuta en dos fases:
  - En la fase de **Mark**, se marcan todas las capas almacenadas en el registro y que están referenciadas por algún manifest. Es decir, se hace una lista de elementos Content-Adressed que **no deben ser eliminados**.

    > *Content-addressed storage (CAS)* es un método para proporcionar acceso rápido a contenido fijo (datos que no se espera que se actualicen) asignándole un lugar permanente en el disco. CAS facilita la recuperación de datos almacenándolos de tal manera que un objeto no se pueda duplicar o modificar una vez que se haya almacenado; por lo tanto, su ubicación es inequívoca

  - En la fase de **Sweep**, se eliminan las capas que no están referenciadas por algún manifest (en la lista creada en el paso

17 https://docs.docker.com/registry/spec/api/#deleting-an-image

anterior).

- El registro debe ponerse en modo **sólo lectura** durante la recolección, ya que en otro caso podrían subirse al registro nuevos manifests durante la fase de *Mark*, referenciando capas que se borran en la segunda fase, y dejando inconsistencias (capas no accesibles).

Más información:

https://docs.docker.com/registry/garbage-collection/

Tags mutables

Si un *tag* (por ejemplo **latest**) se sube múltiples veces al repositorio, pero con distinto manifest, en el registro se mantienen referencias a los manifests tanto por el nombre del tag (latest), como por el Content-Addressed (`@sha256:xxxxxxxx`).

Por tanto, el recolector de basura no será capaz de eliminar las capas referenciadas por *versiones previas* de ese tag, y además no será posible determinar si ese manifest ha dejado de ser usado.

La **API v2 del registro tiene ciertas limitaciones**, como que permite listar los tags de un repositorio, pero no los manifests. Por tanto no es posible, para un determinado tag que ha mutado, determinar los hashes de los manifests previos.

> Un hash es una función que utiliza un algoritmo matemático para transformar un conjunto de datos (por ejemplo, un manifest) en un código alfanumérico con una longitud fija [65]

La eliminación, sin embargo, no se hace por tag, sino por *digest*, lo que permite eliminar una imagen (manifest) específica, pero no un tag completo. Véase:

https://github.com/docker/distribution/pull/2169

https://github.com/docker/distribution/issues/2170

Este comportamiento de la API genera algunos problemas también en Harbor, donde borrar un tag llama a la API usando el *digest* de la imagen, y esto puede producir que otros tags del mismo repositorio, si apuntaban al mismo *digest*, queden inaccesibles y se marquen con tamaño 0 bytes. Véase:

https://github.com/goharbor/harbor/issues/6515

Online Garbage Collection

Docker Enterprise Edition (Docker EE) soporta Online Garbage Collection. Para permitirlo, no utiliza nombres Content-Addressed en el almacenamiento, sino que genera nombres únicos y los enlaces con el Hash en una base de datos.

> Docker Enterprise Edition es la oferta comercial completa del Proyecto Docker, que ofrece características adicionales así como un ciclo extendido de soporte y lanzamiento.[66]

Al hacer la recolección de basura:

1. Establece un *tiempo de corte*
2. Marca cada fichero referenciado por manifest por un tag con una estampa temporal del tiempo de corte. Al *pushear* nuevos archivos al registro, también se marcan con la estampa temporal.
3. *Sweep* (barre) todos los archivos no referenciados que no tienen una estampa temporal posterior al *tiempo de corte* (es decir, referenciados o *pusheados* tras iniciarse el *garbage collection*).
4. Las referencias a los archivos están en una base de datos RethinkDB, por lo que no es necesario leer los manifests para poder detectar si un archivo es referenciado.

Más información:

https://docs.docker.com/ee/dtr/admin/configure/garbage-collection/

CAPÍTULO 10

Seguridad en Docker

Seguridad en Docker

Algunos consejos generales de seguridad en el uso de Docker ya se han ido comentando en apartados anteriores de esta guía. En este capítulo propondremos otros nuevos, de igual importancia y fundamentales para sacarle el máximo partido a Docker y operar con la máxima seguridad. Así, veremos cómo debemos usar Docker, por ejemplo, para protegernos de posibles vulnerabilidades, o cómo auditar los contenedores a través de los *logs* una vez ha habido algún ataque externo o evento, entre otras cosas.

Host

La máquina Host es la parte más importante del entorno Docker, debido a que es la instancia donde se apoya toda la infraestructura y donde se ejecutarán nuestros contenedores. Por esta razón es recomendable seguir unas pautas de buenas prácticas y seguridad a la hora de su configuración:

Partición específica para Docker

El primer punto a tener en cuenta es el particionado del disco duro, debido a que Docker necesitará almacenar las diferentes imágenes que el usuario irá utilizando.

Para ello es imprescindible crear una partición específica para Docker. En Linux es instalada, por defecto, en la ruta `/var/lib/docker`.

Permisos para ejecutar Docker

Una vez instalado Docker en la partición correcta, es necesario indicar qué usuario/s pueden controlar el *demonio* (daemon) de Docker. Este podrá ser controlado por el superusuario del sistema (usuario root, normalmente, el

administrador), y por las personas que se añadan al grupo «docker» (el cual se crea por defecto en el momento de la instalación).

Es recomendable añadir a dicho grupo a aquellos usuarios que deban tener permiso para el control del *demonio* de Docker, en lugar de dar a estos usuarios directamente el acceso a todo el sistema como usuarios root, para así proteger el mismo, y también almacenar de forma segura las credenciales de usuario root.

Otra posibilidad para dar acceso a usuarios al daemon de Docker es utilizar el modo experimental *rootless,* existente desde Docker 19.03 (aunque aún tiene varias limitaciones).

Para más información acerca de esto, consultar: «Run the Docker daemon as a non-root user (Rootless mode)[18]»

En cualquiera de los casos, solo se debería permitir control del daemon Docker a usuarios de confianza. Uno de los principales motivos es que Docker permite compartir un directorio entre el host y un contenedor sin limitaciones de acceso desde el contenedor. Por tanto, se puede crear un contenedor donde la carpeta `/host` sea el / del host, y se puede modificar esta carpeta o incluso hacer un `chroot` desde dentro del contenedor , logrando un acceso completo y sin restricciones al sistema de archivos delhost. Este problema no es específico de Docker, sino que en las máquinas virtuales también es posible dar acceso a un sistema de archivos o incluso directamente a un dispositivo de bloques completo.

Por esta razón Docker no expone su API REST directamente en un socket implementado con protocolo TCP (127.0.0.1), sino que lo hace a través de un socket UNIX, en un archivo que sólo pueden leer/escribir los usuarios del grupo «*docker*» mencionado anteriormente*:*

```
# ls -lh /run/docker.sock
srw-rw---- 1 root docker 0 Jan 19 16:18 /run/docker.sock
```

Si se quiere exponer el socket de Docker a través de la red, debería hacerse **protegido por un firewall.** Aún haciendo esto, el socket será accesible por contenedores o desde la misma máquina, por lo que se debería, además, **activar**

18 https://docs.docker.com/engine/security/rootless/

TLS (HTTPS) y autenticación mediante certificados.

Otra forma de acceder al socket de Docker de forma remota es creando un tunel a través del protocolo SSH (Secure SHell):

```
DOCKER_HOST=ssh://USER@HOST
```

O bien:

```
ssh -L /path/to/docker.sock:/var/run/docker.sock ...
```

Veamos un ejemplo muy sencillo de cómo ser root en un host si tenemos acceso a Docker:

```
$ docker run --rm \
  -v $PWD:/h_docs ubuntu \
  sh -c 'cp /bin/bash /h_docs/ \
  && chmod +s /h_docs/bash'

./bash -p
```

Otra forma, en una línea:

```
$ docker run -it --rm -v /:/host alpine chroot /host
```

Auditoría de ficheros y directorios

Como el *demonio* de Docker se puede ejecutar con privilegios de root, es vital auditar cualquier cambio sobre el mismo. Para ello, habría que activar la auditoría de dichos ficheros, activando un sistema de monitorización que nos alerte de cualquier cambio en los mismos y almacenar un registro de los comandos realizados por los usuarios sobre estos ficheros.

En Linux, por defecto, viene instalado en la mayoría de distribuciones el *demonio* de auditoría Audit que, entre otras cosas, permitirá:

- Auditar accesos y modificaciones de ficheros
- Monitorizar llamadas al sistema
- Detectar intrusiones
- Registrar comandos de los usuarios

Para configurar correctamente el *demonio* de auditoría, será necesario añadir las siguientes reglas al fichero `/etc/audit/rules.d/audit.rules`

- w /usr/bin/docker -k docker
- w /var/lib/docker -k docker
- w /etc/docker -k docker
- w /usr/lib/systemd/system/docker.service -k docker
- w /usr/lib/systemd/system/docker.socket -k docker
- w /etc/default/docker -k docker
- w /etc/docker/daemon.json -k docker
- w /usr/bin/docker-containerd -k docker
- w /usr/bin/docker.runc -k docker

En este caso se trata de los ficheros y directorios por defecto, pero si se realizara alguna modificación de los mismos, sería recomendable añadir estas modificaciones a dicho fichero.

Finalmente, en el fichero `/var/log/audit/audit.log` se podrían revisar los logs generados en la auditoría.

Daemon

El *demonio* de Docker es una parte fundamental, ya que es la pieza encargada de gestionar el ciclo de vida de las distintas imágenes en la máquina host. Esta se encarga tanto de proporcionar recursos como de limitar los mismos.

Limitación del tráfico entre contenedores

Una vez que se está trabajando con contenedores desplegados, Docker, por defecto, permite el tráfico entre los desplegados en el mismo host, por lo tanto puede permitir que cada uno disponga de la capacidad de ver el tráfico del resto de contenedores, ya que se encuentran en la misma red.

Esta posibilidad es una fuente indirecta de divulgación de datos entre los contenedores.

Para limitar esta divulgación existe un comando muy importante que se deberá utilizar siempre que no se desee compartir estos datos. Para acceder al mismo, primero habrá que utilizar el comando `dockerd`, que nos permitirá realizar modificaciones del *demonio*. Después, usando el comando `$ dockerd –icc=false` conseguiremos limitar este tráfico.

Autorización del uso de la CLI

La autorización del uso de la Command Line Interface (CLI) se basa en los grupos de Docker, tal y como se comentaba en el punto Permisos para ejecutar Docker. Por lo tanto, es necesario añadir a este grupo a los usuarios que necesiten tener acceso al control general del *demonio* de Docker y al uso de la CLI.

Persistencia de logs del daemon de Docker

En el mundo de la seguridad, la gestión de los logs es una de las tareas más importantes, ya que permite monitorizar lo que está pasando en todo momento.

En un mismo host se pueden estar ejecutando simultáneamente varios contenedores donde cada uno está generando sus propios ficheros de log.

Para la gestión de los mismos en Docker, existen varios comandos que permiten monitorizar estos logs. En este caso se muestran dos de ellos que permiten obtener la salida convencional de los mismos, `STDOUT` y `STDERR`:

```
$ docker logs <container_id>
$ docker service logs <SERVICE | TASK>
```

En algunos casos no será suficiente con el uso de estos comandos, debido a que no será posible explotar la información mostrada de una forma adecuada. En ese caso se pueden utilizar diferentes opciones que permitan realizar una redirección y un formateo de los mismos. Para ello habría que utilizar el siguiente comando:

```
$ dockerd --log-driver <option>
```

Y disponer así, de las opciones:

Tabla 6: Funcionalidades para la gestion de logs

Opción	Descripción
`none`	El comando `docker logs` no mostrará ninguna salida.
`json-file`	Los logs serán formateados como JSON. Es la opción por defecto si no se especifica ninguna.

Opción	Descripción
`syslog`	Los logs serán formateados como syslog, para su correcto funcionamiento debe de estar ejecutándose el *demonio* de syslog.
`journald`	Los logs serán formateados como journald, para su correcto funcionamiento debe de estar ejecutándose el *demonio* de journald.
`gelf`	Escribe los logs en un endpoint Graylog Extended Log Format (GELF) como podría ser la herramienta Logstash.
`splunk`	Escribe los logs en el software splunk.
`awslogs`	Escribe los logs en el servicio Amazon Cloudwatch Logs
`gcplogs`	Escribe los logs en la plataforma Google Cloud Platform Logging

Un ejemplo sería el siguiente:

```
$ dockerd --log-driver syslog
```

Acceso a ficheros de configuración

Tal y como se ha indicado en anteriores apartados, es muy importante limitar en todo momento los usuarios que disponen de permiso para el control del *demonio* de Docker. Así, además de crear un grupo donde se indiquen dichos usuarios, tendremos que concretar también, para cada fichero o carpeta, el usuario y grupo con permiso sobre la misma y el tipo de permiso, tal y como se muestra en la siguiente tabla [Tabla 7]:

Tabla 7: Acceso a ficheros de configuración

Archivo / Carpeta	Usuario:Grupo	Permiso
`docker.service`	root:root	644 (rw-r--r--)
`docker.socket`	root:root	644 (rw-r--r--)
`/etc/docker`	root:root	755 (rwxr-xr-x)
`daemon.json`	root:root	644 (rw-r--r--)
`/etc/default/docker`	root:root	644 (rw-r--r--)
Certificado del registro de Docker	root:root	440 (r--r-----)
Certificado de la CA de TLS	root:root	440 (r--r-----)
Certificado del servidor de Docker	root:root	440 (r--r-----)
Clave privada del certificado del servidor de Docker	root:root	400 (r--------)

Imágenes a medida

Tal y como se indicó anteriormente (punto Dockerfile de esta guía), podemos crear nuestras propias imágenes a partir de un fichero Dockerfile. A continuación se indican algunas buenas prácticas de seguridad en relación al mismo.

Denegar el uso de root

Se debe especificar el usuario que puede ejecutarse dentro de un contenedor (por defecto, todos lo hacen con el usuario root, lo cual no es una práctica recomendada). Una vez levantado el contenedor, sólo se podrán realizar acciones sobre el mismo con el usuario indicado.

Para ello se deberán añadir los siguientes comandos en el fichero Dockerfile:

```
RUN useradd <options>
USER <username>
```

Veámoslo con un ejemplo. Queremos ejecutar nuestro contenedor con el usuario «Ivan», por lo cual los comandos anteriores deberán de añadirse a nuestro Dockerfile de la siguiente forma:

```
RUN useradd -s /bin/bash ivan
USER ivan
```

De esta forma limitaremos el uso del usuario root y las acciones sólo podrán realizarse con el usuario indicado, «Iván» en este ejemplo.

Verificación de la integridad de la imagen

Otra cosa muy importante de cara a la seguridad es revisar la integridad de una imagen junto con las diferentes capas que la componen. Cada imagen y cada

capa de esta disponen de un hash *SHA256* (código alfanumérico de 256 bits a modo de firma, generado con el algoritmo SHA256) que permite verificar si la imagen descargada ha sido modificada.

Para ello podemos utilizar el comando siguiente:

```
$ docker inspect <image_name>
```

Este comando nos devolverá una salida `json` en la cual se mostrará toda la información de la imagen, entre la que podremos encontrar los hashes con los cuales verificar su integridad.

Sin embargo, si alguna imagen es descargada a través de una red insegura o es publicada por un tercero o posible usuario malicioso, no es posible detectarlo sólo con la información de estos hashes.

Por ello, además de lo ya mencionado, existen dos opciones que se pueden incluir en el fichero Dockerfile para firmar la imagen a la hora de su construcción y al publicar las mismas en un repositorio de Docker. Así, estas podrán ser verificadas criptográficamente y se podrá comprobar la integridad de las mismas de una segunda forma, para mayor seguridad. Será necesario para ello generar los correspondientes pares de claves públicas y privadas para poder realizar las firmas.

Estas opciones son las siguientes:

- `DOCKER_CONTENT_TRUST` → Permite habilitar o deshabilitar la verificación de Content Trust de Docker (DCT), esto es, la utilización de firmas digitales para el envío y recepción de datos. En el caso de estar habilitado, Docker realizará una verificación de la integridad de la misma en el repositorio, ya sea este público o privado, apoyándose en todo momento en el servicio Notary (plataforma de Docker que proporciona la utilidad DCT, encargándose de almacenar y verificar las firmas). Al habilitar DCT (Docker Content Trust), sólo permitiremos descargar y ejecutar imágenes y tags que están firmados digitalmente por *publicadores* conocidos.

- `DOCKER_CONTENT_TRUST_SERVER` → En la mayoría de casos las empresas se nutren de las imágenes del repositorio oficial de Docker, por lo que será este el encargado de verificar la firma de la misma. En el caso contrario de que se utilice un repositorio privado, deberemos especificar nuestro servidor Notary indicando la URL en la que se encuentra el mismo usando esta opción.

Podemos especificar dichas opciones en nuestro fichero Dockerfile, de la siguiente forma:

```
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=”https://notary.docker.io”
```

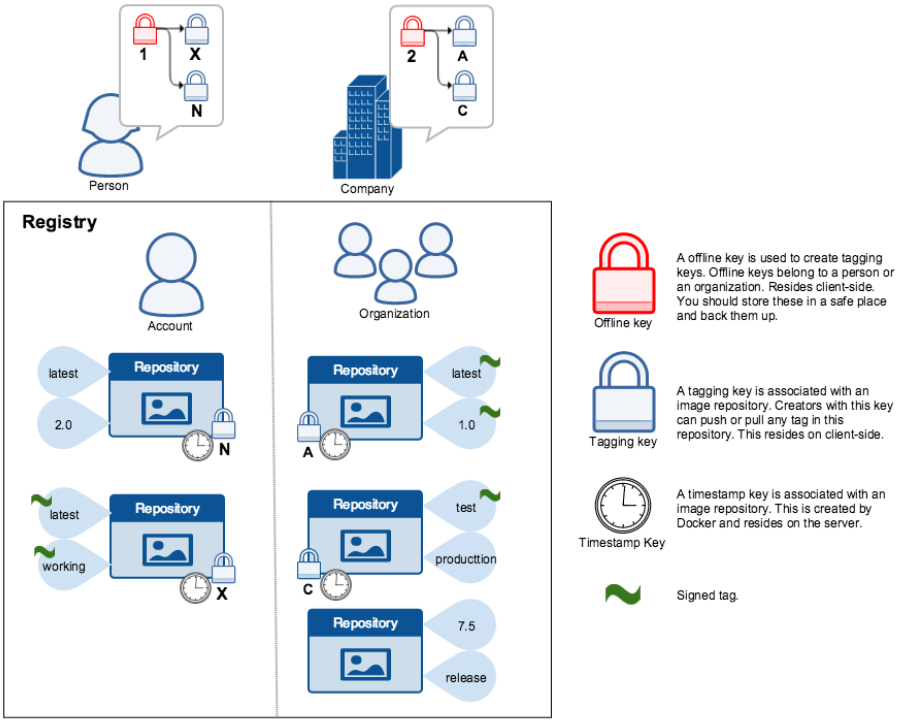

Figura 27: Firmado digital de imágenes

En la siguiente Figura 18 se muestra de forma esquematizada el funcionamiento

del firmado digital de imágenes, así como otros conceptos de interés sobre los que recomendamos profundizar, tales como el «offline key», «tagging key» y «timestamp»:

Para más información a cerca de Content Trust y de la herramienta Notary, se recomienda visitar los siguientes enlaces:

https://docs.docker.com/engine/security/trust/content_trust

https://docs.docker.com/notary/getting_started/

Eliminación de permisos especiales

En Linux existen permisos especiales que normalmente son utilizados para la *granularidad* (esto es, el otorgamiento especial de privilegios) de los accesos a ficheros y directorios del sistema operativo. De esta forma, usuarios que no dispongan de permisos de root, pero si de estos permisos especiales, podrán ser capaces de acceder a ciertos directorios y/o ejecutar archivos concretos. Se trata de los permisos **setuid** (set user ID) y **setgid** (set group ID).

Pues bien, en Docker es posible especificar en el fichero Dockerfile, con la instrucción RUN, un comando que elimine estos permisos especiales evitando posibles vulnerabilidades en los mismos.

El comando sería el siguiente:

```
RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true
```

Este comando realizará una búsqueda de los ficheros ejecutables y eliminará cualquier permiso especial de los mismos, evitando con ello el uso de estos por usuarios que no deban tener esa capacidad.

Evitar el almacenamiento de credenciales

Cuando se genera una imagen mediante un fichero Dockerfile, es muy común que al declarar diferentes credenciales para poder realizar pruebas de conexión, estas no se eliminen correctamente una vez concluidas las pruebas.

La solución, en este caso, reside en revisar los *secretos* que se están exponiendo (véase apartado Credenciales [secrets] página 183) y delegar esta funcionalidad a orquestadores como podría ser Docker Swarm o Kubernetes.

Se puede encontrar más información al respecto en la documentación oficial de los mismos, en los siguientes enlaces:

- Docker Swarm

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

- Kubernetes

  https://kubernetes.io/docs/concepts/configuration/secret

Permisos y restricciones en contenedores

Restricción de permisos

Por defecto, los contenedores arrancan con una serie de privilegios limitados. Uno de los puntos fuertes que ofrecen los mecanismos de seguridad de Docker es la granularidad a la hora de proporcionar estos privilegios en caso de que la situación lo requiera, pudiendo llevar a cabo, así, la ejecución sin necesidad de proporcionar permisos de root.

Para ello, Docker proporciona el siguiente comando:

```
$ docker run <comando>
```

Mediante el cual podemos añadir o eliminar privilegios con las siguientes opciones respectivamente:

```
$ docker run --cap-add={option}
$ docker run --cap-drop={option}
```

Entre las posibles opciones a añadir o eliminar, encontramos las siguientes:

Tabla 8: Privilegios que pueden ser añadidos o eliminados de contenedores.

Opción	Descripción
`SETPCAP`	Si un fichero no dispone de capacidades, las añade al conjunto de permitidas, o las elimina.
`MKNOD`	Crea ficheros especiales usando el comando `mknod`.

Opción	Descripción
`AUDIT_WRITE`	Escribe registros al log de auditoria del kernel.
`CHOWN`	Permite modificar el propietario y el grupo de un fichero.
`NET_RAW`	Permite el uso del RAW socket (que opera sin utilizar un protocolo de comunicación en concreto y permiten la posibilidad de protocolos de capa 3 y 4) y del PACKET socket (utilizado para enviar y recibir datos a través de un RAW socket). Permite el uso de herramientas como ping (para acceder a un contenedor en ejecución desde otro contenedor usando simplemente el nombre de Docker en lugar de una dirección IP) o tcpdump (para análisis de tráfico en la red en tiempo real).
`DAC_OVERRIDE`	Bypass (esquivar sistema de seguridad) en las comprobaciones de los permisos de lectura, escritura y ejecución de un fichero.
`FOWNER`	Bypass en las comprobaciones de los permisos que requiere que el UID del proceso coincida con el UID del fichero.
`FSETID`	No modifica los permisos `setuid` y `setgid` cuando se modifica un fichero
`KILL`	Bypass en las comprobaciones de permisos para el envío de señales.
`SETGID`	Permite añadir un mapeo GID a un namespace de usuario.

Restricción de contenedores privilegiados

Siguiendo con el tema del punto anterior, en este apartado se trata la posibilidad de ejecutar un contenedor con privilegios especiales.

Una de las funcionalidades donde por defecto no se dispone de privilegios es en la ejecución de Docker dentro de Docker (véase *Docker-in-docker,*)

Para poder añadir esta funcionalidad se deberá añadir el flag siguiente al comando `docker run`:

```
$ docker run --privileged <image_name>
```

Es vital indicar que este tipo de acciones no debería de utilizarse a no ser que la situación lo requiera, dado que, aplicando la mentalidad de buenas prácticas de seguridad, es necesario ser minimalista en todo momento y lo más restrictivo posible.

Restricción de protocolos y puertos

Tal y como se comentaba en puntos anteriores es muy importante la seguridad de nuestros contenedores, y para ello, entre otras cosas, es vital permitir sólo aquellos puertos y protocolos que vayan a ser utilizados.

Por una parte, con respecto a la restricción de protocolos, es importante entender que el uso del protocolo SSH no debería de estar permitido en ninguno de los contenedores que se desplieguen. Este protocolo sólo debe de estar permitido en la máquina Host, desde la cual se controlarán los contenedores, ya que la comunicación con estos se realizará por medio del *demonio* de Docker.

La solución a esta medida (si utilizamos una imagen base oficial de un sistema operativo concreto, y no una imagen para un servicio específico) es la desinstalación de protocolo SSH.

Para ello se deberán lanzar los siguientes comandos:

- En distribuciones Red Hat, CentOS o Fedora se utilizará uno de los siguientes, en función de la versión utilizada de la misma:

```
$ sudo dnf remove openssh-server
$ sudo yum remove openssh-server
```

- Y en el caso de distribuciones basadas en el sistema operativo Debian se utilizará este otro:

```
$ sudo apt-get --purge remove openssh-server
```

Por otro lado, con respecto a la restricción de puertos, Docker, por defecto, utilizará un puerto disponible dentro del rango 49153 - 65535 (en caso de que no se especifique un puerto concreto).

Una buena práctica de seguridad consistiría en no permitir que los contenedores pudiesen mapear los puertos privilegiados (1-2014) de la máquina Host. Pero existen excepciones, como podría ser el caso de un contenedor que esté ejecutando un servicio HTTP o HTTPS: en este caso no habría ningún problema en mapear el puerto 80 o 443, respectivamente, de la instancia Host al contenedor.

Limitación de recursos

Por defecto, los contenedores comparten los recursos de la máquina host de forma equitativa, esto quiere decir que no existe ninguna preferencia entre los diferentes contenedores a la hora de consumir recursos.

En ocasiones es posible que una aplicación requiera consumir más recursos que otra. Para ello, Docker dispone de un comando que permite aumentar o limitar estos recursos en una imagen concreta:

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

Este comando dispone de varios parámetros de configuración que permiten limitar o priorizar el consumo de recursos en caso de que una aplicación lo requiera.

Para visualizar las diferentes opciones de configuración disponibles se puede lanzar el siguiente comando:

```
$ docker run --help | egrep "cpu|device|memory"
```

El cual mostrará una salida por pantalla como la expuesta en la siguiente imagen [Figura 28]:

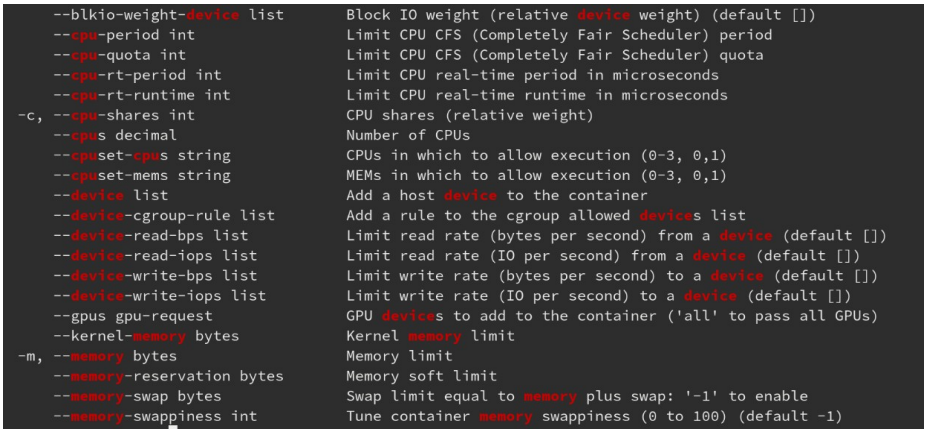

Figura 28: Salida mostrada por pantalla tras la ejecución del anterior para la configuración del consumo de recursos en Docker

Bench Security

Debido a la importancia que tiene la seguridad en el campo de la informática, Docker cuenta con una herramienta llamada Docker Bench Security que, junto con el resto de recomendaciones que hemos ido exponiendo hasta ahora, garantiza las mejores condiciones de seguridad en la utilización de Docker.

Docker Bench Security consiste en un script que ejecuta y revisa cada una de las buenas prácticas de seguridad recomendadas e indica si realmente se encuentran implantadas y el estado de las mismas, antes de empezar a trabajar con nuestro entorno Docker en entornos en producción.

Todas estas buenas prácticas, están inspiradas en el documento de referencia del Center for Internet Security **(CIS) Docker Community Edition (CE) Benchmark**, el cual proporciona una guía de las mismas y de su aplicación en los diferentes entornos, tal y como se vieron en los puntos anteriores (máquina Host, *demonio* de Docker, prácticas de seguridad en imágenes y contenedores, etcétera).

Este documento se puede descargar desde su repositorio en GitHub, en la siguiente URL:

https://github.com/docker/docker-bench-security

Para realizar una comprobación de seguridad utilizando esta herramienta, solo tendremos que descargarnos el repositorio de GitHub:

- Git clone

https://github.com/docker/docker-bench-security.git

Y posteriormente ejecutar el script:

```
cd docker-bench-security
sudo ./docker-bench-security.sh
```

Tras realizar la instalación por defecto y lanzar el script, se muestran los diferentes errores de seguridad que han sido encontrados, tal y como vemos en la siguiente captura Figura 29:

```
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.3.5
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Benchmark v1.2.0.
# ------------------------------------------------------------------------------

Initializing Wed Jan 22 10:57:10 UTC 2020


[INFO] 1 - Host Configuration

[INFO] 1.1 - General Configuration
[NOTE] 1.1.1  - Ensure the container host has been Hardened
[INFO] 1.1.2  - Ensure Docker is up to date
[INFO]        * Using 19.03.5, verify is it up to date as deemed necessary
[INFO]        * Your operating system vendor may provide support and security maintenance for Docker

[INFO] 1.2 - Linux Hosts Specific Configuration
[WARN] 1.2.1 - Ensure a separate partition for containers has been created
[INFO] 1.2.2  - Ensure only trusted users are allowed to control Docker daemon
[INFO]        * docker:x:999:vagrant
[WARN] 1.2.3  - Ensure auditing is configured for the Docker daemon
[WARN] 1.2.4  - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.2.5  - Ensure auditing is configured for Docker files and directories - /etc/docker
[WARN] 1.2.6  - Ensure auditing is configured for Docker files and directories - docker.service
[WARN] 1.2.7  - Ensure auditing is configured for Docker files and directories - docker.socket
[WARN] 1.2.8  - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO] 1.2.9  - Ensure auditing is configured for Docker files and directories - /etc/sysconfig/docker
[INFO]        * File not found
[INFO] 1.2.10  - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
[INFO]         * File not found
[WARN] 1.2.11  - Ensure auditing is configured for Docker files and directories - /usr/bin/containerd
[INFO] 1.2.12  - Ensure auditing is configured for Docker files and directories - /usr/sbin/runc
[INFO]         * File not found


[INFO] 2 - Docker daemon configuration
[WARN] 2.1  - Ensure network traffic is restricted between containers on the default bridge
[PASS] 2.2  - Ensure the logging level is set to 'info'
[PASS] 2.3  - Ensure Docker is allowed to make changes to iptables
```

Figura 29: Salida mostrada por pantalla tras realizar la comprobación de seguridad, en la que se indican los diferentes errores de seguridad encontrados.

Análisis estático de vulnerabilidades

El análisis estático de vulnerabilidades (estático, ya que se realiza sin necesidad de ejecutar el programa) es una medida que actualmente se está llevando a cabo cada vez más en el ámbito de la seguridad informática. Este análisis se integra en todo el ciclo de vida del proyecto (durante sus diferentes fases de desarrollo) para que pueda, así, aportar información y solventar los posibles problemas que existan antes de que el proyecto salga a producción.

Además, esta acción es vital para evitar cualquier problema generado por un atacante externo a nuestra infraestructura. Este sería el único análisis realizado que caería dentro de las etapas incluidas en los *pipelines* de entrega y despliegue continuo (CI/CD), y debe ser un nuevo requisito a nivel de seguridad, tanto en la generación de imágenes como de contenedores en los diferentes entornos durante la integración continua.

Cabe mencionar que, aunque a nivel de Dockerfile se realicen buenas prácticas, no es hasta este momento cuando se podría verificar qué librerías y binarios se han incluido en la propia imagen, y donde realmente se comprueba que se han seguido correctamente todas estas pautas. Además, es la propiedad de inmutabilidad de las imágenes Docker la que hace tan importante este análisis estático de vulnerabilidades: esta propiedad establece que, una vez creada la imagen, esta no puede ser modificada y, en caso de ser necesaria su modificación, sería obligatorio volver a construirla.

Una de las herramientas existentes que permite realizar un análisis de las imágenes Docker y de sus vulnerabilidades es **Clair**. Esta examina los contenidos de las imágenes y busca si los paquetes de software contienen vulnerabilidades registradas en una base de datos de CVEs (Common Vulnerabilities and Exposures).

Otra herramienta que también realiza escaneo de vulnerabilidades, así como detección de malas prácticas en la construcción de contenedores, es **Anchore** Engine (esta se encuentra dentro de un paquete comercial ofrecido por Anchore).

Anchore es una solución Open Source, disponible en versión de servicio proporcionado por terceros, o en versión On-Premise (local), cuyo principal objetivo es el descubrimiento de vulnerabilidades conocidas en imágenes, ya sea en repositorios públicos o privados.

Esta solución permite obtener el listado de paquetes instalados en el sistema operativo de la imagen Docker y realizar un análisis de los mismos con respecto a las vulnerabilidades conocidas, con el fin de solventar todas o la gran mayoría de ellas.

Con el fin de ver cómo se realiza este análisis de vulnerabilidades con la herramienta anchore, se propone a continuación una práctica, en la cual se escaneará una imagen creada previamente con un fichero Dockerfile y subida a nuestro repositorio en DockerHub.

Antes de proceder al análisis de nuestra imagen se deberán de realizar ciertas acciones:

El primer paso será levantar el entorno necesario:

```
docker-compose up -d
```

Posteriormente accederemos al contenedor denominado como “api”:

```
docker exec -it <container-id> /bin/bash
```

Y sobre este contenedor crearemos dos variables para indicar el usuario del administrador, en este caso, y la URL en la cual dispone de su servidor.

```
export ANCHORE_CLI_USER=admin
export ANCHORE_CLI_URL=http://<ip>:8228/v1
```

Completado este paso tendremos dos posibilidades para realizar el análisis: acceder al contenedor y utilizar directamente el cliente de anchore (opción A), o ejecutarlo desde fuera mediante el cliente de Docker (opción B):

A) En el caso de hacerlo directamente desde el contenedor:

Ver el estado de anchore:

```
anchore-cli system status
```

Ver las imágenes disponibles:

```
anchore-cli image list
```

Añadir una imagen nueva:

```
anchore-cli image add docker.io/library/debian
```

Comprobar el estado de análisis de una imagen. Mostrará un refresco de su estado cada 5 segundos:

```
anchore-cli image wait docker.io/library/debian
```

Y por último, ver las vulnerabilidades descubiertas en una imagen:

```
anchore-cli image vuln <image> os | non-os | all
```

B) En el caso de hacerlo desde el cliente de Docker:

Ver el estado del sistema:

```
docker-compose exec engine-api anchore-cli system status
```

Ver las imágenes disponibles:

```
docker-compose exec engine-api anchore-cli image list
```

Añadir una imagen nueva:

```
docker-compose exec engine-api  anchore-cli image add
docker.io/library/debian
```

Comprobar el estado de análisis de una imagen. Mostrará un refresco de su estado cada 5 segundos:

```
docker-compose exec engine-api  anchore-cli image wait
docker.io/library/debian
```

Y por último, ver las vulnerabilidades descubiertas en una imagen:

```
docker-compose exec engine-api  anchore-cli image vuln <image> os |
non-os | all
```

A continuación se muestra un ejemplo de la imagen analizada del sistema operativo Debian, donde se muestran las vulnerabilidades encontradas en el mismo, así como el paquete al cual pertenecen y un enlace a la documentación de dicha vulnerabilidad:

```
[anchore@64400620f1c7 anchore-engine]$ anchore-cli image vuln docker.io/library/debian all
Vulnerability ID        Package                         Severity        Fix         CVE Refs        Vulnerability URL
CVE-2005-2541           tar-1.30+dfsg-6                 Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2005-2541
CVE-2007-5686           login-1:4.5-1.1                 Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2007-5686
CVE-2007-5686           passwd-1:4.5-1.1                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2007-5686
CVE-2010-4051           libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4051
CVE-2010-4051           libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4051
CVE-2010-4052           libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4052
CVE-2010-4052           libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4052
CVE-2010-4756           libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4756
CVE-2010-4756           libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2010-4756
CVE-2011-3374           apt-1.8.2                       Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2011-3374
CVE-2011-3374           libapt-pkg5.0-1.8.2             Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2011-3374
CVE-2011-3389           libgnutls30-3.6.7-4+deb10u2     Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2011-3389
CVE-2011-4116           perl-base-5.28.1-6              Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2011-4116
CVE-2012-2663           libxtables12-1.8.2-4            Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2012-2663
CVE-2013-4235           login-1:4.5-1.1                 Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2013-4235
CVE-2013-4235           passwd-1:4.5-1.1                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2013-4235
CVE-2013-4392           libsystemd0-241-7~deb10u3       Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2013-4392
CVE-2013-4392           libudev1-241-7~deb10u3          Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2013-4392
CVE-2017-11164          libpcre3-2:8.39-12              Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2017-11164
CVE-2017-16231          libpcre3-2:8.39-12              Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2017-16231
CVE-2017-18018          coreutils-8.30-3                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2017-18018
CVE-2017-7245           libpcre3-2:8.39-12              Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2017-7245
CVE-2017-7246           libpcre3-2:8.39-12              Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2017-7246
CVE-2018-1000654        libtasn1-6-4.13-3               Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2018-1000654
CVE-2018-20796          libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2018-20796
CVE-2018-20796          libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2018-20796
CVE-2018-6829           libgcrypt20-1.8.4-5             Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2018-6829
CVE-2019-1010022        libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2019-1010022
CVE-2019-1010022        libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2019-1010022
CVE-2019-1010023        libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2019-1010023
CVE-2019-1010023        libc6-2.28-10                   Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2019-1010023
CVE-2019-1010024        libc-bin-2.28-10                Negligible      None                        https://security-tracker.debian.org/tracker/CVE-2019-1010024
```

Figura 30: Ejemplo del resultado de un análisis estático de vulnerabilidades realizado sobre una imagen de Debian

Anchore ofrece, además, una utilidad para hacer el escaneo de vulnerabilidades en línea:

```
$ curl -sO https://ci-tools.anchore.io/inline_scan-v0.6.0
$ chmod +x inline_scan-v0.6.0
$ ./inline_scan-v0.6.0 -r airadier/test:latest
...
```

Nos generará una carpeta `anchore-reports/` con una serie de documentos json.

Anchore permite, además, definir una serie de políticas en formato JSON para determinar cuándo una imagen es aceptada o no, o qué reglas y buenas prácticas se deben aplicar.

Podemos encontrar ejemplos en Anchore Hub y gestionarlas mediante la herramienta de Anchore CLI:

https://github.com/anchore/hub/tree/master/sources/bundles

Por otra parte, la herramienta de seguridad Sysdig Secure ofrece también un servicio de escaneo de imágenes, tanto en repositorio como inline, y un interfaz gráfico para definir las políticas y mostrar los resultados [Figura 31 y Figura 32]:

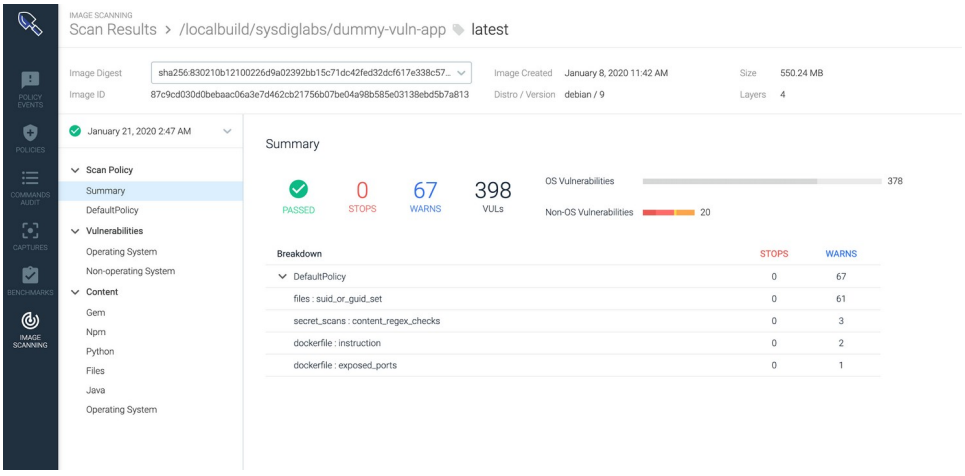

Figura 31: Escaneo de imágenes mediante la herramienta Sysdig

Una de las ventajas de Sysdig es que el análisis de la imagen (que es la parte más costosa) se realiza una única vez. Por lo tanto, si cambiamos las políticas, sólo necesitaremos reevaluar los metadatos de la imagen con las nuevas políticas para que el resultado del escaneo se actualice instantáneamente.

Orquestadores como Kubernetes ofrecen, además, mecanismos de *Admission Control,* que se pueden integrar con estas herramientas de escaneo de imágenes, de forma que podamos impedir la ejecución de contenedores que no cumplan ciertos requisitos de seguridad.

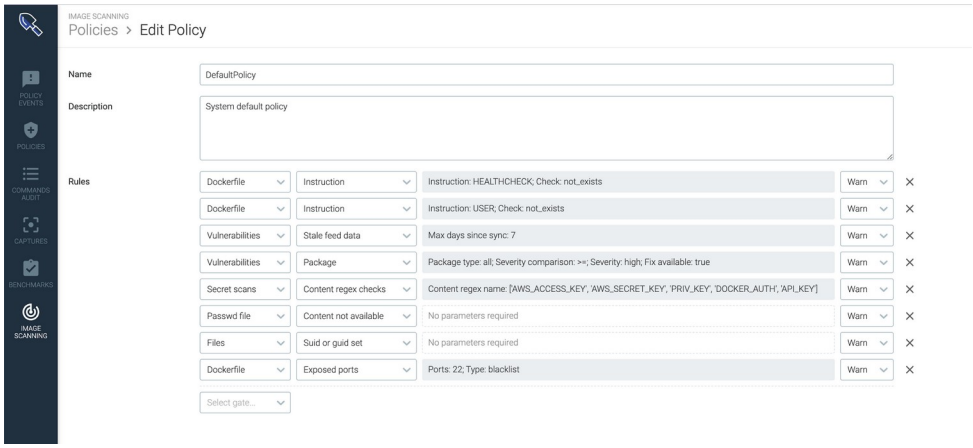

Figura 32: Edición de políticas para la ejecución de escaneo de imágenes mediante la herramienta Sysdig

Más información y recursos:

- Anchore Engine Inline Scanning | Enterprise Documentation

https://docs.anchore.com/current/docs/engine/usage/integration/ci_cd/inline_scanning/

- Cloud Native + Kubernetes image scanning

https://sysdig.com/products/kubernetes-security/image-scanning/

https://github.com/sysdiglabs/secure-inline-scan

Mecanismos adicionales de seguridad

Además de los vistos hasta ahora, existen mecanismos adicionales de seguridad que es importante conocer y aplicar:

Seccomp

Secure Computing (`seccomp`) es una característica adicional del kernel de Linux que Docker también puede aprovechar para restringir las acciones que un contenedor puede realizar.

```
$ docker info

…
 Security Options:
  apparmor
  seccomp
   Profile: default
…
```

Por defecto, el uso de `seccomp` impide realizar cualquier llamada al sistema, excepto *`read`*, *`write`*, *`_exit`* y *`sigreturn`*, aunque pueden configurarse diferentes perfiles. El perfil *default* que utiliza Docker deshabilita el acceso a unas 44 llamadas de un total de más de 300 y puede encontrarse aquí:

https://github.com/moby/moby/blob/master/profiles/seccomp/default.json

Para definir y utilizar un perfil distinto:

```
$ docker run --rm -it \
    --security-opt seccomp=/path/to/seccomp/profile.json \
    hello-world
```

Y para deshabilitar el confinamiento de llamadas al sistema:

```
$ docker run --rm -it \
    --security-opt seccomp=unconfined \
    hello-world
```

Más información:

https://docs.docker.com/engine/security/seccomp/

AppArmor

AppArmor es un módulo de seguridad del kernel de Linux que proporciona control de acceso para confinar programas a un conjunto limitado de recursos, protegiéndose así ante amenazas de seguridad.

Docker crea un perfil AppArmor llamado «`docker-default`» y lo carga en el kernel. Es un perfil de protección moderada, a la vez que proporciona la máxima compatibilidad de aplicaciones. La plantilla a partir de la cual se crea el perfil se puede encontrar en:

https://github.com/moby/moby/blob/master/profiles/apparmor/template.go

Se pueden definir perfiles de *enforcement* (impiden la acción y además reportan el intento), o de *compliance* (únicamente se registra el acceso), y también se puede modificar el perfil utilizado con un contenedor utilizando la opción `--security-opt` de Docker:

```
$ apparmor_parser -r -W /path/to/your_profile
$ docker run --rm -it --security-opt apparmor=your_profile hello-world
```

Más información:

https://wiki.ubuntu.com/AppArmor

https://docs.docker.com/engine/security/apparmor/

Contenedores privilegiados y *capabilities*

Por defecto, los contenedores Docker se ejecutan en modo no-privilegiado, y por lo tanto hay ciertas acciones que no pueden realizar. Por ejemplo, no se puede ejecutar un Docker daemon dentro de un contenedor, ya que no se tiene acceso a los dispositivos especiales.

Sin embargo, para este tipo de necesidades especiales, existe la posibilidad de ejecutar un contenedor en modo *privilegiado* (asumiendo el riesgo de seguridad que esto conlleva), de la siguiente manera:

```
$ docker run --privileged --rm -it hello-world
```

O se puede proporcionar acceso a algunos dispositivos concretos:

```
$ docker run --device=/dev/snd:/dev/snd:w ...
```

El flag "`:w`" proporciona acceso de lectura, al igual que "`:r`", y "`:m`" permite hacer `mknod`. También se pueden combinar, por ejemplo para proporcionar acceso de lectura, escritura y mknod usando "`:rwm`"

Además, también se podrá añadir o denegar el acceso a ciertas *capabilities* (privilegios). Por ejemplo, para permitir todo tipo de acciones privilegiadas excepto `MKNOD`:

```
$ docker run --cap-add=ALL --cap-drop=MKNOD ...
```

Por defecto, algunas *capabilities* son permitidas:

- `MKNOD`: comando que permite crear ficheros especiales de bloques o caracteres [67]
- `CHOWN` : Permite modificar el propietario y el grupo de un fichero
- `KILL`: Bypass en las comprobaciones de permisos para el envío de señales
- `NET_BIND_SERVICE`: Permite enlazar el socket de un servicio a un puerto

privilegiado, esto es, puerto de red < 1024)

Y otras denegadas:

- `SYS_MODULE`: Permite la modificación sin restricciones del kernel, por ejemplo, cargar y eliminar módulos del mismo.
- `SYS_RAWIO`: Permite que los contenedores envíen comandos SCSI a los discos desde dentro de los contenedores
- `SYS_TIME`: Permite establecer la hora del sistema y el bloqueo en tiempo real.
- `SYS_PTRACE`: Permite hacer trazas a cualquier proceso con la llamada al sistema `ptrace`

Se puede ver la lista completa de *capabilities* en el siguiente enlace:

http://man7.org/linux/man-pages/man7/capabilities.7.html

O ampliar la información a cerca de privilegios y capabilities en la documentación del comando `run`:

https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities

CAPÍTULO 12

Ejemplos prácticos

Ejemplos prácticos

Para finalizar esta guía, expondremos en este último apartado una serie de cuestiones y ejemplos prácticos que es importante conocer para terminar de dominar el mundo Docker. Recomendamos seguir las explicaciones y realizar los ejercicios indicados para poner asentar las mismas sobre la práctica, junto con otras cuestiones que hemos ido viendo de una manera más teórica en el transcurso de esta guía.

Entornos de desarrollo

En el siguiente ejercicio se propone la descarga y ejecución de algunos de los contenedores mostrados en la siguiente lista, los cuales permiten a un desarrollador disponer de un entorno de desarrollo completo y reproducible para distintos lenguajes de programación. Este tipo de contenedores son muy utilizados para procesos de integración continua, dentro de un Pipeline, pero también pueden ser usados localmente para homogeneizar los entornos de compilación de los desarrolladores, de forma que todos ellos construyan la aplicación utilizando un entorno idéntico, misma versión, y sin factores externos que puedan modificar el resultado.

Para ello, se puede reemplazar lo que era una instalación de un compilador y herramientas de *build* tipo Maven, Gradle, etc. por un contenedor en el que se monta la carpeta con el código fuente, y se ejecuta el proceso de compilación, copiando los artefactos resultantes fuera del contenedor, o en una nueva imagen que será la utilizada para el despliegue.

PARTE 1

Se deben descargar y ejecutar los siguientes contenedores:

➔ Go: https://hub.docker.com/_/golang

Ejemplo:

https://github.com/callicoder/go-docker

➔ OpenJDK https://hub.docker.com/_/openjdk *

➔ Maven https://hub.docker.com/_/maven*

https://github.com/jenkins-docs/simple-java-maven-app

➔ Gradle: https://hub.docker.com/_/gradle

Ejemplo:

https://github.com/jabedhasan21/java-hello-world-with-gradle

➔ NodeJS: https://hub.docker.com/_/node/

Ejemplo:

https://github.com/adamhalasz/docker-node-example

Podemos clonar el repositorio y ejecutar el ejemplo simplemente con:

- `docker build --tag docker-node-example-image`
- `docker run --rm -p 8080:9000 docker-node-example-image`
- `Abrir localhost:8080 en el navegador`

PARTE 2

Desarrollo directamente dentro del contenedor con Visual Studio Code:

https://code.visualstudio.com/docs/remote/containers

- git clone https://github.com/Microsoft/vscode-remote-try-node
- git clone https://github.com/Microsoft/vscode-remote-try-python
- git clone https://github.com/Microsoft/vscode-remote-try-go
- git clone https://github.com/Microsoft/vscode-remote-try-java
- git clone https://github.com/Microsoft/vscode-remote-try-dotnetcore
- git clone https://github.com/Microsoft/vscode-remote-try-php
- git clone https://github.com/Microsoft/vscode-remote-try-rust
- git clone https://github.com/Microsoft/vscode-remote-try-cpp

PARTE 3

Clonar el repositorio de Java y probar algunos ejemplos: ejecutar, breakpoint, etc.

Pipelines de CI/CD

A continuación se propone realizar las siguientes prácticas, que se describirán en la página web que muestra cada enlace, a cerca de los *pipelines* de integración y distribución continua:

➔ Jenkins:

https://jenkins.io/doc/book/pipeline/docker/

➔ Azure Pipelines Container Jobs:

https://docs.microsoft.com/en-us/azure/devops/pipelines/process/container-phases?view=azure-devops

➔ Github actions:

1. https://github.com/features/actions
2. https://help.github.com/en/actions/building-actions/creating-a-docker-container-action

➔ Tekton: CI/CD nativo para Kubernetes -

https://cloud.google.com/tekton

Contenedores reutilizables / parametrizables

Casi cualquier herramienta o aplicación común dispone de su versión *dockerizada*, ya sea una imagen oficial, o creada por terceros. A continuación, se muestra una lista con las más importantes, sobre las que se propone trabajar a modo de práctica, así como sobre la configuración por variables de entorno [] y por bind-mount []

➔ **Ngnix:**

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

```
docker run --name some-nginx -p 8080:80 -v $PWD:/usr/share/nginx/html:ro -d nginx
```

Simplemente con este comando ejecutaremos un servidor HTTP Nginx sirviendo el contenido de la carpeta actual ($PWD)

➔ **Apache HTTP:**

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

➔ **PostgreSQL:**

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

- Levantar con el docker-compose de ejemplo
- Activar persistencia
- Ver parámetros, variables de entorno, etc.
- Ver cómo *adminer* se conecta a *db* por el nombre del contenedor

➔ **MariaDB:**

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

- Ejemplo similar con *adminer*

➔ **WordPress:**

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

- `docker run --name some-wordpress -p 8080:80 -d wordpress`
- ¡Ojo! Nos falta BD
- Usar `docker-compose.yml` del ejemplo

➔ **Jenkins:**

https://hub.docker.com/r/jenkins/jenkins

Docs: https://github.com/jenkinsci/docker/blob/master/README.md

- `docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts`
- Ver *admin* password en salida del log (stdout)

Más ejemplos:

A) https://hub.docker.com/_/sonarqube

B) https://hub.docker.com/_/vault

C) https://hub.docker.com/_/haproxy

D) https://hub.docker.com/_/elasticsearch

E) https://hub.docker.com/_/kibana

F) https://hub.docker.com/_/mysql

G) https://hub.docker.com/_/registry

H) https://hub.docker.com/_/rabbitmq

Configuración por variables de entorno

Con el fin de facilitar la configuración y reutilización de contenedores, es común que se permita establecer valores de configuración simples mediante variables de entorno.

A la hora de crear nuestra propia aplicación o contenedor, la aplicación ejecutada debe tener en cuenta estas variables de entorno. En otro caso, para soportar configuración por variables de entorno necesitaremos crear un script *wrapper*, y usarlo como el *entrypoint* del contenedor. Este script deberá procesar estas variables de entorno y convertirlas en los correspondientes archivos de configuración usados por la aplicación, o en los los correspondientes parámetros al lanzar el proceso principal de la aplicación.

Podemos especificar variables de entorno al ejecutar un contenedor de distintas maneras:

- `docker -e VARIABLE_NAME=value …`
- docker-compose.yml, usando sección *environment* o `env_file:`

  https://docs.docker.com/compose/environment-variables/

- Archivo **.env**

Configuración por bind-mount

La opción **`-v`** de Docker, o la sección *volumes* nos permiten hacer *`bind mount`* de un **fichero** o de una **carpeta** del host dentro de un contenedor. De esta manera podemos reemplazar toda una carpeta (normalmente lo usaremos para proveer de *persistencia* al contenedor), o también un **archivo**, pudiendo *inyectar* archivos de configuración dentro del contenedor, reemplazar archivos por defecto, crear nuevos archivos (carpetas `conf.d/` o similares), etc.

Paquetización de utilidades

Un uso muy común de contenedores es empaquetar utilidades sencillas junto a todas sus dependencias, de forma que su utilización sea sencilla, sin tener que proceder a la instalación local de la aplicación y todas sus dependencias.

Por ejemplo, la compañía Hashicorp, que provee imágenes de distintas versiones del software Terraform, permitiendo usar versiones antiguas (por algunos cambios o providers soportados en 0.11.x y 0.12.x)

- Terraform:

  https://hub.docker.com/r/hashicorp/terraform

Despliegue de app compleja usando Docker Compose

Se propone realizar el ejemplo mostrado en el siguiente enlace sobre una aplicación Node/MongoDB *dockerizada*:

https://github.com/bradtraversy/docker-node-mongo

- Cambiar para exponer en puerto 8080 y hacer accesible desde el host
- Activar persistencia (`docker-compose.yml`, añadir *volumes* y mapear `./mongo-data` a `/data/db`)

Véase también la siguiente información sobre Laradock, un entorno completo de desarrollo PHP basado en Docker:

- https://laradock.io/
- https://github.com/laradock/laradock

REFERENCIAS

Bibliografía

1: Maxlinux.es, Todo lo que quisiste saber del comando "sudo", 2019, https://maslinux.es/todo-lo-que-quisiste-saber-del-comando-sudo/

2: Icot.es, Introducción a los contenedores, , https://www.icot.es/introduccion-a-los-containers/

3: Autores de Wikipedia, Virtualización, , https://es.wikipedia.org/wiki/Virtualizaci%C3%B3n#cite_ref-PH-19_1-0

4: Citrix.com, ¿Qué es la virtualización de aplicaciones?, , https://www.citrix.-com/es-es/glossary/what-is-application-virtualization.html

5: Institut Puig Castellar, Introducción a los grupos de control (cgroups) de Linux, , https://elpuig.xeill.net/Members/vcarceler/articulos/introduccion-a-los-grupos-de-control-cgroups-de-linux

6: Autores de Wikipedia, Docker (software), 2020, https://es.wikipedia.org/wiki/Docker_(software)

7: Sistemas.com, Definición Host, , https://sistemas.com/host.php

8: Autores de Wikipedia, Tiempo de ejecución, , https://es.wikipedia.org/wiki/Tiempo_de_ejecuci%C3%B3n

9: EltallerdelBit.com, Imágenes de Docker, 2018, https://eltallerdelbit.com/imagenes-docker/

10: Digital Guide Ionos, Tutorial de Docker: instalar y gestionar la plataforma de contenedores, , https://www.ionos.es/digitalguide/servidores/configuracion/tutorial-docker-instalacion-y-primeros-pasos/

11: Microsoft Azure, Qué es PasS, , https://azure.microsoft.com/es-es/overview/what-is-paas/

12: Shubheksha Jalan, An Introduction To Docker Tags, 2018, https://dev.to/shubheksha/introduction-of-docker-tags-51gn

13: Deyimar A., Variables de entorno de Linux: cómo leerlas y configurarlas en un VPS de Linux, 2019, https://www.hostinger.es/tutoriales/variables-de-entorno-linux-como-leerlas-y-configurarlas-vps/

14: Carlos Villagómez, Variables de entorno, 2017, https://es.ccm.net/contents/652-variables-de-entorno

15: Linuxize.com, How to Set and List Environment Variables in Linux, 2019, https://linuxize.com/post/how-to-set-and-list-environment-variables-in-linux/

16: SpeedCheck.com, Puerto, , https://www.speedcheck.org/es/wiki/puerto/#fn1

17: Porjcrug, Listado de puertos TCP/IP. Cambiar el puerto de escucha para Escritorio remoto Windows Server, , https://www.tele-pc.es/listado-de-puertos-tcp-ip-cambiar-el-puerto-de-escucha-para-escritorio-remoto-windows-server/

18: Docker, Use volumes, , https://docs.docker.com/storage/volumes/

19: Kimberly Luna, Crea tu primer comando en Bash, 2017, https://medium.com/techwomenc/crea-tu-primer-comando-en-bash-aa80eeffe2c7

20: Docker Docs, Docker overview, , https://docs.docker.com/get-started/overview/

21: Techopedia.com, Socket, 2011, https://www.techopedia.com/definition/16208/socket

22: NeoWiki, Concepto Api de Rest, , https://neoattack.com/neowiki/api-de-rest/

23: Qloudea.com, ¿Qué es TCP/IP?, 2017, https://soporte.qloudea.com/hc/es/articles/115003659065--Qu%C3%A9-es-TCP-IP-

24: Autores de Wikipedia, Protocolo de control de transmisión, , https://es.wikipedia.org/wiki/Protocolo_de_control_de_transmisi%C3%B3n

25: Avi, Autor en Geekflare.com, How to change Docker sock file location?,

2019, https://geekflare.com/change-docker-sock-file-path/

26: Autores de Wikipedia, Open Container Initiative, , https://en.wikipedia.org/wiki/Open_Container_Initiative

27: w3schools.com, JSON - Introduction, , https://www.w3schools.com/js/js_json_intro.asp

28: Michael Kerrisk, Linux Programmer's Manual, , http://man7.org/linux/man-pages/man2/uname.2.html

29: Autores de Wikipedia, Linux namespaces, , https://en.wikipedia.org/wiki/Linux_namespaces#UTS

30: Docker Docs, Use bind mounts, , https://docs.docker.com/storage/bind-mounts/

31: Autores de Wikipedia, Dynamic frequency scaling, , https://en.wikipedia.org/wiki/Dynamic_frequency_scaling#cite_note-1

32: Autores de Wikipedia, Completely Fair Scheduler, , https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

33: Michael Kerrisk, The Linux Programming Interface, 2010, http://man7.org/linux/man-pages/man7/cpuset.7.html

34: Sylabs Inc, Limiting container resources with cgroups, 2019, https://sylabs.io/guides/3.0/user-guide/cgroups.html

35: Michael Kerrisk, The Linux Programming Interface, , http://man7.org/linux/man-pages/man2/ptrace.2.html

36: Autores de Wikipedia, Union mount, , https://en.wikipedia.org/wiki/Union_mount

37: SoftwareUsco, Discos virtuales, , https://compusoftwareusco.webnode.com.co/novedades/software/discos-virtuales/

38: RedHat.com, Sistema de archivos XFS, , https://access.redhat.com/docu-

mentation/es-es/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-storage-xfs

39: Docker, Use the OverlayFS storage driver, , https://docs.docker.com/storage/storagedriver/overlayfs-driver/

40: Docker, Use the BTRFS storage driver, , https://docs.docker.com/storage/storagedriver/btrfs-driver/

41: Wiki Kernel, Glossary Btrsf, , https://btrfs.wiki.kernel.org/index.php/Glossary

42: Docker, Use the ZFS storage driver, , https://docs.docker.com/storage/storagedriver/zfs-driver/

43: Docker, Understand images, containers, and storage drivers, , https://testdockerrr.readthedocs.io/en/latest/userguide/storagedriver/imagesandcontainers/

44: Jeff Hale, 15 Docker Commands You Should Know, 2019, https://towardsdatascience.com/15-docker-commands-you-should-know-970ea5203421

45: Sara, forera en desarrolloweb.com, Qué hace el comando PWD en linux?, , https://desarrolloweb.com/faq/que-hace-comando-pwd

46: Docker, Network Overview, , https://docs.docker.com/network/

47: Sahiti Kappagantula autor en edureka.co, Docker Networking – Explore How Containers Communicate With Each Other, 2019, https://www.edureka.co/blog/docker-networking/

48: Autores de Wikipedia, Loopback, , https://es.wikipedia.org/wiki/Loopback

49: linuxito.com, Tutorial básico de iptables en Linux, 2016, https://www.linuxito.com/seguridad/793-tutorial-basico-de-iptables-en-linux

50: Nigel Brown autor en windsock.io, The docker-proxy, 2015, https://windsock.io/the-docker-proxy/

51: Digital Guide IONOS, El servidor DNS y la resolución de nombres en Internet, 2019, https://www.ionos.es/digitalguide/servidores/know-how/que-es-el-ser-

vidor-dns-y-como-funciona/

52: Jason Gunthorpe, Guía de usuario de APT, 1998, https://www.debian.org/doc/manuals/apt-guide/ch2.es.html

53: itToolbox.com, What is wrapper scripts?, 2011, https://it.toolbox.com/question/what-is-wrapper-scripts-062411

54: Glosario IT, Descripción "pipeline", , https://www.glosarioit.com/Pipeline

55: Redhat.com, ¿Qué es la automatización?, , https://www.redhat.com/es/topics/automation/whats-it-automation

56: Jiang Huan, Docker build cache sharing on multi-hosts with BuildKit and buildx, 2019, https://pracucci.com/graceful-shutdown-of-kubernetes-pods.html

57: Tugurium Glosario Terminologia Informatica, Definición footprint, , http://www.tugurium.com/gti/termino.php?Tr=footprint

58: GlosarioIT.com, Definición de Buffer, , https://www.glosarioit.com/Buffer

59: Jack Wallen, What is the difference between Dockerfile and docker-compose.yml files?, 2019, https://www.techrepublic.com/article/what-is-the-difference-between-dockerfile-and-docker-compose-yml-files/

60: BBVA, Gestión de secretos en contenedores Docker, 2017, https://www.bbva.com/es/gestion-secretos-contenedores-docker/

61: Autores de Wikipedia, Palabra (informática), , https://es.wikipedia.org/wiki/Palabra_(inform%C3%A1tica)

62: Sreenivas Makam's Blog, Docker in Docker and play-with-docker, 2016, https://sreeninet.wordpress.com/2016/12/23/docker-in-docker-and-play-with-docker/

63: Platzi, Multi-tenant: Qué es y por qué es importante, , https://platzi.com/blog/multi-tenant-que-es-y-por-que-es-importante/

64: Github.com, Concepts, , https://github.com/RequirementEngineering/hello-

world-al159949/wiki/Concepts

65: Gonzalo García-Valdecasas, ¿Qué son las funciones Hash y para que se utilizan?, 2018, https://www.cysae.com/funciones-hash-cadena-bloques-block-chain/

66: hoplasoftware.com, ¿Por que elegir Docker EE frente a CE?, , Ref.: https://hoplasoftware.com/por-que-elegir-docker-ee-frente-a-ce/

67: Ubuntu Manpage Repository, bionic (1) mknod.1.gz, 2019, http://manpages.ubuntu.com/manpages/bionic/es/man1/mknod.1.html

Enlaces