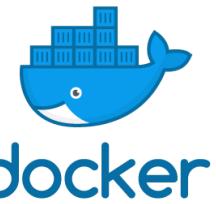




POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



Análisis de contenedores Docker

- y sus implicaciones de seguridad

Javier Alonso Silva

Seguridad en Sistemas y Redes

Universidad Politécnica de Madrid

2021



Resumen

TO-DO

Índice

1. Introducción	1
1.1. ¿Qué es Docker?	3
1.2. <i>Real-life usages</i>	7
1.3. <i>Docker rules</i>	10
2. Docker	13
2.1. Estructura de un Docker	13
2.2. Creación de un contenedor	22
2.3. Comunicación entre contenedores	24
2.4. Despliegue de aplicaciones multi-contenedores. <i>docker-compose</i>	29
2.5. “Orquestación” de contenedores	35
2.5.1. Herramientas de orquestación	37
2.6. Líneas futuras de desarrollo e innovación	48
3. Seguridad en Docker	49
3.1. Análisis de la pila Docker	49
3.2. Diferencias fundamentales con <i>chroot</i>	55
3.3. Seguridad en las comunicaciones de red – <i>firewall</i>	55
3.4. Conclusiones	56
Referencias	57

1. Introducción

La era tecnológica ha avanzado en los últimos años a pasos agigantados, y las demandas del sector han crecido junto a ella. No hace más de 200 años se “descubría” la electricidad; hace 90 años nacía la primera computadora básica capaz de realizar operaciones aritméticas; hace 70 años nacía el transistor que sustituyó las válvulas de vacío (figura 1); y desde entonces, el crecimiento ha sido exponencial [1].



Figura 1: Comparativa de una válvula de vacío (izquierda) frente a un transistor (centro) y un circuito integrado (derecha).

Otro de los ejemplos de tecnologías que han crecido exponencialmente son los dispositivos de almacenamiento, donde no hacía más de 20 años las capacidades máximas se estimaban en torno a los MB (megabytes) y ahora se hablan de EB (exabytes) [2]. Esta evolución es muy representativa también a nivel económico, ya que el coste del almacenamiento ha ido bajando a medida pasaba el tiempo, así como el espacio físico que ocupan los dispositivos (figura 2):

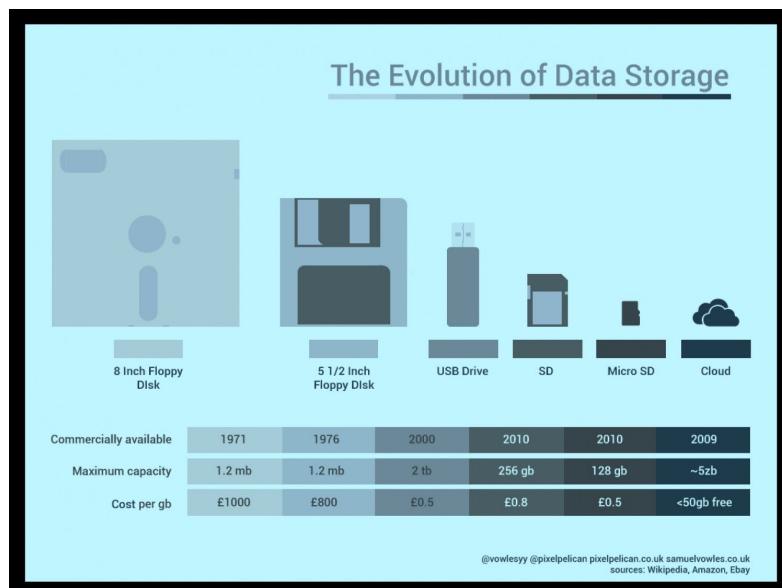


Figura 2: Evolución del espacio de almacenamiento en términos económicos y cuantitativos [3].

Finalmente, el gran salto tecnológico se ha producido con la aparición de Internet y las comunicaciones ya no eran únicamente personales sino entre dispositivos. En relación con el punto anterior, la aparición de Internet ha permitido descentralizar el espacio donde ya el usuario no guarda su información en su equipo personal sino en un clúster de servidores distribuidos a nivel mundial al cual accede, de forma simultánea, desde Internet y desde cualquier dispositivo. Así, lo que comenzó como una red de conexión de unos pocos usuarios ha acabado convirtiéndose en la red global que todos usamos y que conecta más de 4 billones de dispositivos (figura 3).

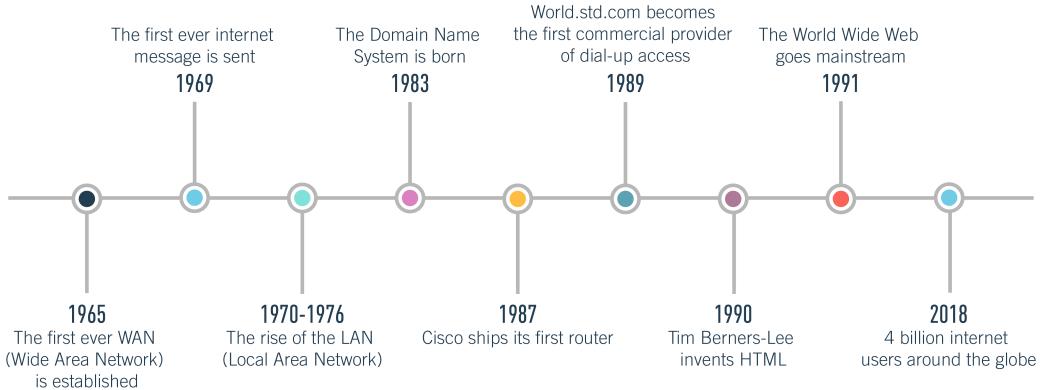


Figura 3: Evolución de Internet a lo largo del tiempo, hasta llegar a hoy [4].

El problema a esto es evidente: con una mayor capacidad de cómputo, con más opciones de comunicación y con más posibilidad de almacenar datos, los requisitos de las aplicaciones van creciendo y creciendo y cada vez son más complejos de satisfacer, no necesariamente a nivel *hardware* (que por lo general suele acompañar) sino a nivel *software*. Como las aplicaciones se orientan a los usuarios es necesario añadir capas de abstracción (como el sistema operativo) para facilitar la labor a la persona. Sin embargo, cada capa nueva que se añade dificulta las tareas de despliegue y mantenimiento dado que existe una gran variedad de combinaciones *hardware* y cada una puede estar con un sistema operativo distinto.

Por otra parte, la extensión de dependencias y posible incompatibilidad entre ellas suele desembocar en el uso de versiones desactualizadas de una librería ya que tendríamos “paquetes rotos”. Esto es tan común que tiene hasta su propio término coloquial “*dependency hell*” [5]. Contar con dependencias obsoletas que ya han cumplido con su ciclo de vida *software* conlleva unas implicaciones de seguridad bastante severas:

- Si un *software* no ha mejorado a lo largo del tiempo, existe una malicia humana que puede aprovecharse de distintos *exploits* existentes y comprometan nuestra aplicación.
- Un *software* no actualizado puede tener implicaciones directas sobre el sistema en que se ejecuta, pudiendo producir fallos en el mismo. Esto se debe principalmente a que el *hardware* sigue mejorando y creciendo y un *software* antiguo puede presentar *bugs* en dispositivos modernos que no presentaría en antiguos.

- Un *software* no actualizado puede comprometer otros elementos del sistema en que se ejecuta. Por ejemplo, una aplicación ‘A’ hace uso de dicho *software* y una aplicación ‘B’ también. Sin embargo, la última aplicación se ha diseñado para trabajar con la última versión del *software* pero la aplicación ‘A’ solo puede funcionar con una versión antigua e insegura. Por consiguiente, pese a que la aplicación ‘B’ funcionaría correctamente el hecho de usar una versión antigua e insegura del *software* compromete directamente al sistema y a la aplicación.

Es por eso que existen alternativas como “*chroot*” y máquinas virtuales para subsanar estos problemas. Sin embargo, en los últimos años ha aparecido una herramienta muy sonada y con gran éxito: Docker y los contenedores.

1.1. ¿Qué es Docker?

Docker es una plataforma abierta diseñada para el desarrollo, despliegue y ejecución de aplicaciones [6]. La idea fundamental que reside detrás de Docker es la de separar la infraestructura de las aplicaciones de manera que se pueda entregar el *software* rápidamente.

Por debajo, Docker ofrece una plataforma que otorga la habilidad de empaquetar y ejecutar las aplicaciones en un entorno aislado llamado “contenedor” (*container*). Entre otras características, un contenedor permite ejecutar una aplicación de forma segura sobre el host en cuestión. La pregunta que surge es, ¿qué es un contenedor?

Contenedores

Un contenedor es una unidad estándar *software* que empaqueta código y todas sus dependencias de manera que la aplicación se ejecuta rápidamente y de forma fiable bajo múltiples entornos de ejecución [7]. Una imagen Docker es un paquete ligero, independiente y ejecutable que incluye absolutamente todo lo necesario para poder ejecutar una aplicación: desde el código en sí hasta el *runtime*, herramientas del sistema, bibliotecas y configuraciones.

Durante la ejecución, una imagen se convierte en un contenedor que se ejecuta sobre la máquina Docker (*Docker Engine*), la cual se encuentra disponible en entornos Linux y Windows.

Al fin y al cabo, los contenedores nos aseguran que una aplicación que hemos desarrollado se va a ejecutar de la misma manera en una máquina u otra. El uso del motor Docker permite ejecutar múltiples contenedores sobre un mismo anfitrión sin añadir demasiada carga en el sistema e indiferentemente de la infraestructura que exista por debajo (figura 4):

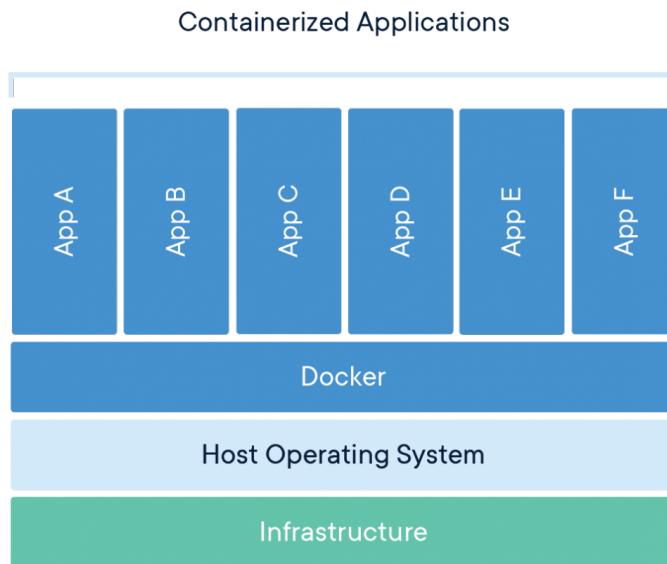


Figura 4: Distribución de los contenedores sobre el motor de ejecución de Docker [7].

La distribución de los contenedores mostrada en la figura 4 puede parecerse mucho a la distribución que tendríamos en una máquina virtual. Sin embargo, hay varias características que lo distinguen principalmente:

1. Un contenedor se ejecuta directamente sobre la máquina anfitriona, mientras que una máquina virtual requiere de un hipervisor.
2. Un contenedor es una abstracción de la capa de aplicación que encapsula el código y las dependencias juntas, mientras que una máquina virtual es una abstracción de una capa física *hardware*.
3. Un contenedor comparte el kernel con el sistema operativo anfitrión, por lo que tiene un gran rendimiento; mientras, una máquina virtual ejecutará su propio kernel sobre el hipervisor del sistema operativo anfitrión.
4. El espacio que necesita un contenedor es muy pequeño en comparación con el de una máquina virtual, que engloba y encapsula un sistema operativo al completo.

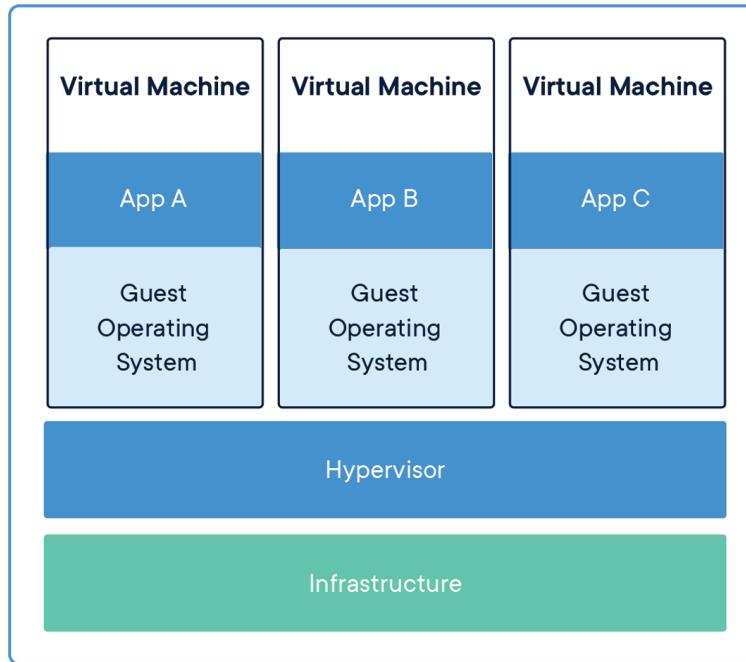


Figura 5: Capas de abstracción de una máquina virtual sobre una máquina anfitriona [7].

En la figura 5 se puede apreciar cómo una máquina virtual añade muchas más capas de abstracción que ralentizan el rendimiento. Sin embargo, esto no quiere decir que sean una mala alternativa: la realidad es que se combinan las dos para obtener una gran flexibilidad para desplegar aplicaciones – contenedores cuando se quiere ejecutar algo directamente sobre el anfitrión; máquinas virtuales para emular *hardware* y que ejecuten en su interior contenedores para ejecutar aplicaciones fácilmente.

La evolución y constante mantenimiento de los contenedores ha generado lo que se conoce como estándar de la industria “*containerd*”. Este estándar define claramente qué arquitectura debe tener un contenedor por debajo y está en constante evolución a medida que la industria crece y madura.

Además, la especificación anterior ha pasado de ser un mero estándar a una aplicación en sí de gestión y orquestación de contenedores, permitiendo que aplicaciones distintas de Docker hagan uso de la arquitectura basada en contenedores aprovechando la OCI: *Open Container Initiative*

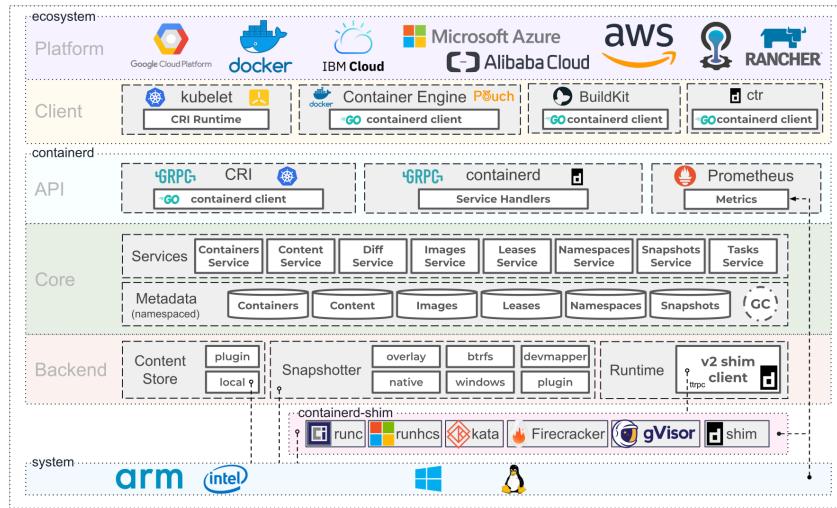


Figura 6: Entorno de ejecución de *containerd* basado en *runC* de la OCI [8].

Docker Engine

El motor de ejecución de Docker establece la arquitectura de ejecución *de facto* que es utilizable desde distintas distribuciones Linux y servidores Windows [9].

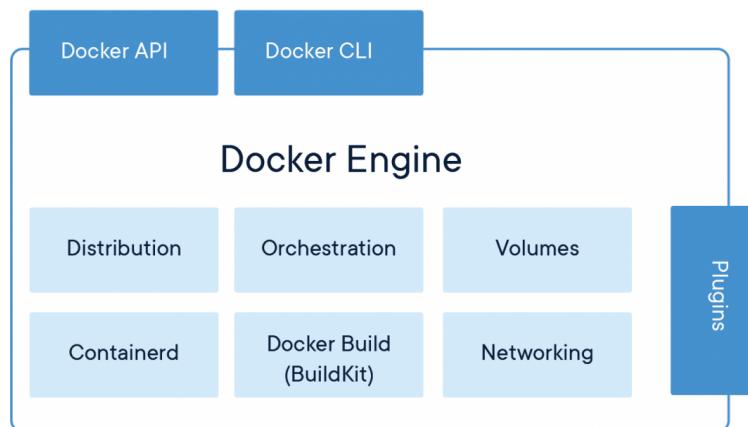


Figura 7: Arquitectura del entorno de ejecución de Docker [9].

El motor de ejecución de Docker se compone de una gran cantidad de elementos que encapsulan de forma uniforme multitud de aptitudes de un sistema operativo o una aplicación (figura 7). Este entorno de ejecución sin embargo es complejo ya que engloba multitud de elementos físicos, como pueden ser las interfaces de red y los volúmenes.

Esto resulta fundamental ya que los contenedores Docker no tienen ni que confiar en la

red del anfitrión: tienen su propio *stack* de red para realizar las comunicaciones que necesiten. Con el motor de ejecución de Docker se busca solventar esos problemas “*dependency hell*” que se han comentado anteriormente y la situación de “en mi equipo funciona”.

De los elementos mostrados en la figura 7, se tiene que son:

- *Distribution*: la distribución Linux en la que se basa el contenedor. Actualmente, Docker solo permite ejecutar contenedores basados en Linux.
- *Orchestration*: cuando hay múltiples contenedores, la orquestación es el proceso por el cual el motor de ejecución de Docker gestiona y maneja qué contenedores se ejecutan, cómo se comunican, cuáles hay que crear nuevos y cuáles eliminar. Es de las partes más complejas que existen en el mundo de los contenedores y ha evolucionado a clústers mucho más completos (y complejos) como Kubernetes o Docker Swarm.
- *Volumes*: los volúmenes (conjuntos de datos) que se manejan en los contenedores. Debido a su arquitectura cerrada, los datos que genera un contenedor solo están visibles para ese contenedor mientras este esté en ejecución. Cuando finalice, todos los datos no persistentes son eliminados.
- *Containerd*: el estándar y cliente de ejecución y manejo de los contenedores a muy bajo nivel.
- *Docker Build (BuildKit)*: herramienta de libre distribución que transforma los ficheros *Dockerfile* en imágenes Docker, listas para ser usadas y distribuidas.
- *Networking*: *stack* de red completo que se pone a disposición de cada contenedor Docker. Cada aplicación puede crear su propio dispositivo de red que cumpla con los requisitos que necesita. Existen varios tipos de adaptadores: *bridge*, *NAT* y *host*. El primero se emplea para realizar comunicaciones a través de red entre distintos contenedores; el segundo para realizar comunicaciones con el exterior mediante una conexión de red completamente independiente a la del anfitrión; la tercera para compartir la interfaz de red del anfitrión con el contenedor, como si fuese una aplicación interna.

Con todo lo anterior, una aplicación puede ejecutarse muy fácilmente en cualquier equipo que integre el motor de ejecución de Docker.

1.2. *Real-life usages*

Desde que nació en 2013, Docker ha ido creciendo y con ello las aplicaciones directas que ha encontrado en el mercado.

Sandboxing

Una de las principales ventajas que otorgó Docker desde su nacimiento fue el de aislar las aplicaciones entre sí y, por consiguiente, ofrecer un entorno de “caja de arena” (*sandbox*) en donde ejecutar nuestras aplicaciones (o aplicaciones inseguras) con cierta confianza [10].

Es cierto que esta característica ya estaba asentada con las máquinas virtuales, y funcionaba correctamente y de forma efectiva. Sin embargo, el tiempo de despliegue y lentitud de una máquina virtual hacía que usarlas para este propósito fuese costoso y no resultase interesante.

Con los contenedores se puede tener un entorno aislado que funciona igual de rápido que una aplicación nativa con unos tiempos de despliegue y uso de recursos limitado. Como se ha visto anteriormente, el motor de ejecución de Docker tiene el control sobre un montón de componentes virtuales y reales que permiten, entre otros, limitar y restringir el acceso a los recursos *hardware* del dispositivo. Bajo esta premisa, un contenedor puede usar solo cierta cantidad de CPU, RAM o red y que cambie en tiempo de ejecución.

Portabilidad

Otra de las características fundamentales de los contenedores es la capacidad de encapsular un *software* y todas sus dependencias. Esto convierte a los contenedores en una gran solución portable: sabemos que si una aplicación funciona en Docker en un equipo Linux, funcionará en Docker de otro equipo Linux exactamente igual, sin necesidad de realizar ningún cambio e indiferentemente de la distribución.

Con la llegada de WSL2, el kernel de Linux se introdujo al completo dentro de las máquinas Windows 10, permitiendo que Docker se pudiera ejecutar de “forma nativa”[11]. Con esto, la limitación anterior se elimina y los contenedores diseñados para Linux funcionarán también en Windows.

Esto ha tenido una repercusión directa con el auge de los sistemas basados en la nube, los cuales a veces resultaban complejos y tediosos. Con los contenedores, una aplicación que un desarrollador ejecuta *on-premise* en su equipo puede ser fácilmente desplegada a un entorno *cloud* sin necesidad de preocuparse si cumple los requisitos o instala las dependencias. La única restricción es que el entorno *cloud* al que se mueva soporte Docker.

Arquitectura de composición

Una gran mayoría de aplicaciones que se ejecutan actualmente están ejecutándose sobre una pila de aplicaciones: servidor web, base de datos, caché en memoria, gestión de logs, etc. La pregunta es, ¿qué sucedería si se encapsula cada una de esas aplicaciones en un contenedor?

Así nace la arquitectura de microservicios, tan popular y estandarizada hoy en día. Un microservicio define un elemento único de una aplicación (que puede ser usado entre 1...n veces) el cual acelera y facilita las labores de desarrollo de una aplicación. Entre otras ventajas, un microservicio puede ser actualizado, reemplazado, eliminado o modificado sin afectar al resto de microservicios que componen una aplicación. Esta alternativa se ha asentado como la solución ideal a las aplicaciones monolíticas monstruosas, que lo engloban todo (como XAMPP) ya que han demostrado ser mucho más fáciles de mantener y desarrollar.

Escalado y orquestación

Aprovechando la arquitectura de microservicios y contenedores, existen técnicas de escalado y orquestación automáticas basadas en Docker y contenedores.

De esta manera, en picos de conexión se despliegan automáticamente más contenedores que gestionan entre ellos las peticiones entrantes y salientes. Cuando las solicitudes bajan, los contenedores en desuso desaparecen para dejar de usar recursos.

Entre las herramientas más sonadas para la gestión de contenedores está Kubernetes, desarrollado por Google. La idea de orquestación nace a raíz de esta empresa que empieza a invertir cantidades millonarias de dinero en contenedores porque le ve un nuevo potencial: las comunicaciones vía Internet de los contenedores. Hasta ahora solo hemos visto un modelo de arquitectura: un cliente Docker que ejecuta uno o varios contenedores. Sin embargo, con la aparición de los microservicios y la orquestación, y dadas las características de red de los contenedores, se abre la posibilidad de que múltiples clientes Docker en máquinas físicamente distintas puedan estar ejecutándose de forma simultánea y compartiendo datos entre ellos fácilmente.

Debido a las capas de aislamiento de Docker, esta comunicación no es sencilla: no sirve con comunicar dos direcciones IP. Sin embargo, utilizando un motor de Docker distribuído se pueden realizar las conexiones como si de una LAN se tratase, cuando en realidad se está usando una red *overlay*. Esto se muestra en la figura 8:

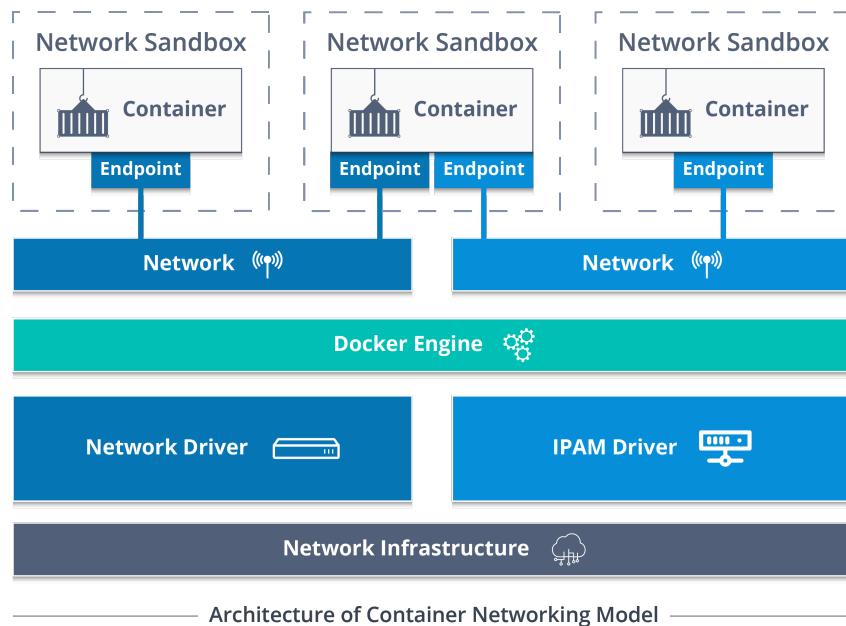


Figura 8: Comunicación entre contenedores usando el motor de Docker [12].

Con todas estas ideas en mente, es evidente que Docker ofrece soluciones fáciles y sencillas para escalar automáticamente aplicaciones alrededor de un clúster de nodos distribuído por el mundo.

Muchas de las aplicaciones de Docker surgen en el mundo DevOps, en donde la mayoría de herramientas de integración continua (CI) y despliegue continuo (CD) han migrado sus infraestructuras hacia Docker. Con esto se consigue que los tests y las compilaciones se hagan de forma sencilla con contenedores de un solo uso.

Otra aplicación directa son las arquitecturas *cloud*, en donde antes había que configurar cientos de parámetros para desplegar una aplicación web basada en PHP y MySQL y ahora

basta con usar uno o varios contenedores que agrupen las funcionalidades que necesitamos.

Por otra parte, gracias a los contenedores los tiempos de desarrollo se han agilizado mucho. Antes, por ejemplo, una aplicación requería de compilar ciertos paquetes y realizar ciertas instalaciones que llevaban mucho tiempo. Con Docker, se exponen las librerías necesarias y se trabaja directamente con aquello que se necesita, sin necesidad de dedicar tiempo a esas tareas.

1.3. Docker rules

Desde su nacimiento en 2013 hasta su expansión mundial hace poco más de 4 años, en 2017/18, los contenedores se han convertido en el *modus operandi* de muchas empresas, que han visto en la tecnología de contenedores una gran ventaja y forma de despegar y aumentar su producción.

Desde entonces, diversos estudios como el llevado a cabo por Portworx cada año brindan la oportunidad de ver qué tecnologías dominan el mercado y cómo va evolucionando el mundo de los contenedores.

De entre los datos obtenidos, es destacable la adopción de contenedores en las empresas tecnológicas: un 87 % de los encuestados (2019) afirman usar contenedores en comparación con el 55 % registrado en 2017. Es más, el 90 % de las aplicaciones que ejecutan en esos contenedores están en entornos de producción, una gran diferencia con 2018 (84 %) y 2017 (67 %) [13].

Estos datos radican en la inversión económica que las empresas realizan en labores de “contenerización”, invirtiendo entre \$500 000 y \$1 000 000 [13]. De entre todos los motivos que mueven a las empresas a realizar esas inversiones, prima la seguridad de los datos sobre los demás.

Parece ser que una de las principales labores de los contenedores en estas decisiones es la de proteger la información (61 %), gestionar las vulnerabilidades fácilmente (43 %) y proteger el sistema en tiempo de ejecución (34 %). Estos datos van directamente ligados con las medidas de seguridad que las compañías adoptan al usar contenedores:

- Cifrar los datos (64 %).
- Monitorización en tiempo de ejecución (49 %).
- Escaneo de vulnerabilidades en los registros de contenedores (49 %).
- Escaneo de vulnerabilidades en las operaciones de CI/CD (49 %).
- Bloquear anomalías mediante la protección en tiempo de ejecución (48 %).

El siguiente motivo de la gran adopción de contenedores es que agiliza mucho la velocidad en el desarrollo y la eficiencia. Por otra parte, la portabilidad de los contenedores permite a las empresas poder mover sus entornos de producción y desarrollo entre una y otra plataforma de nube públicas, de entre las cuales las más usadas (12 % de la muestra) son AWS, Azure y Google Cloud [13].

En particular, se observa cómo AWS (la plataforma de Amazon) es la dominante en este sector, llevándose el 78 % del sector; la siguiente, Azure con el 39 %; y finalmente, GCP (*Google Cloud Platform*) con el 35 % y subiendo rápidamente [14]. Destaca el crecimiento de Google ya que es quien empezó a invertir mucho dinero en contenedores desde su nacimiento y el creador de Kubernetes, la tecnología de orquestación más usada a nivel mundial.

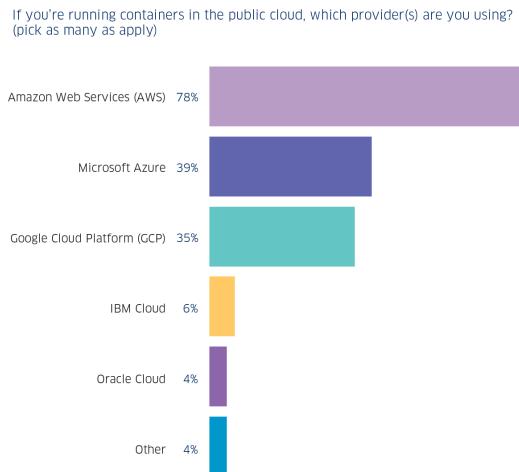


Figura 9: Uso de contenedores según la plataforma *cloud* [14].

La situación mencionada anteriormente se ve directamente reflejada en la “contenerización” de aplicaciones en según que plataforma. De los usuarios de Azure, solo el 20 % ha creado un contenedor para más de la mitad de sus aplicaciones, significativamente más bajo que el 33 % de los no usuarios. Esto se ve drásticamente reducido cuando se hablan de aplicaciones en entorno de producción [14].

Por el contrario, casi un tercio de los usuarios de GCP (31 %) han creado un contenedor para más de la mitad de sus aplicaciones, relativamente superior al 27 % de los no usuarios. Este mismo efecto se produce con respecto a las aplicaciones en producción desplegadas en GCP [14].

Esto se ve reflejado en el gráfico de la figura 10:

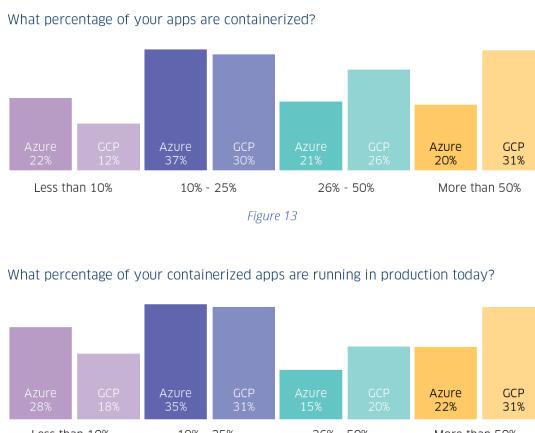


Figura 10: Porcentaje de las aplicaciones desplegadas en contenedores en según qué plataformas [14].

En un estudio más moderno, se estima en el año 2020 ha supuesto un mayor auge en las tecnologías de “contenerización”, en donde los responsables de IT han priorizado la creación de contenedores para aplicaciones ya existentes, migrar toda la infraestructura a la nube y hacer un mejor uso de las plataformas en la nube. De entre todos los problemas, el principal es cumplir con los requisitos legales, de rendimiento y regulatorios vigentes según las necesidades de la industria; y la portabilidad de las aplicaciones, las cuales estaban confinadas y diseñadas para sistemas en particular y ahora se quieren desplegar en la nube en general [15].

Esto se ve en la infografía diseñada por Forrester (figura 11):

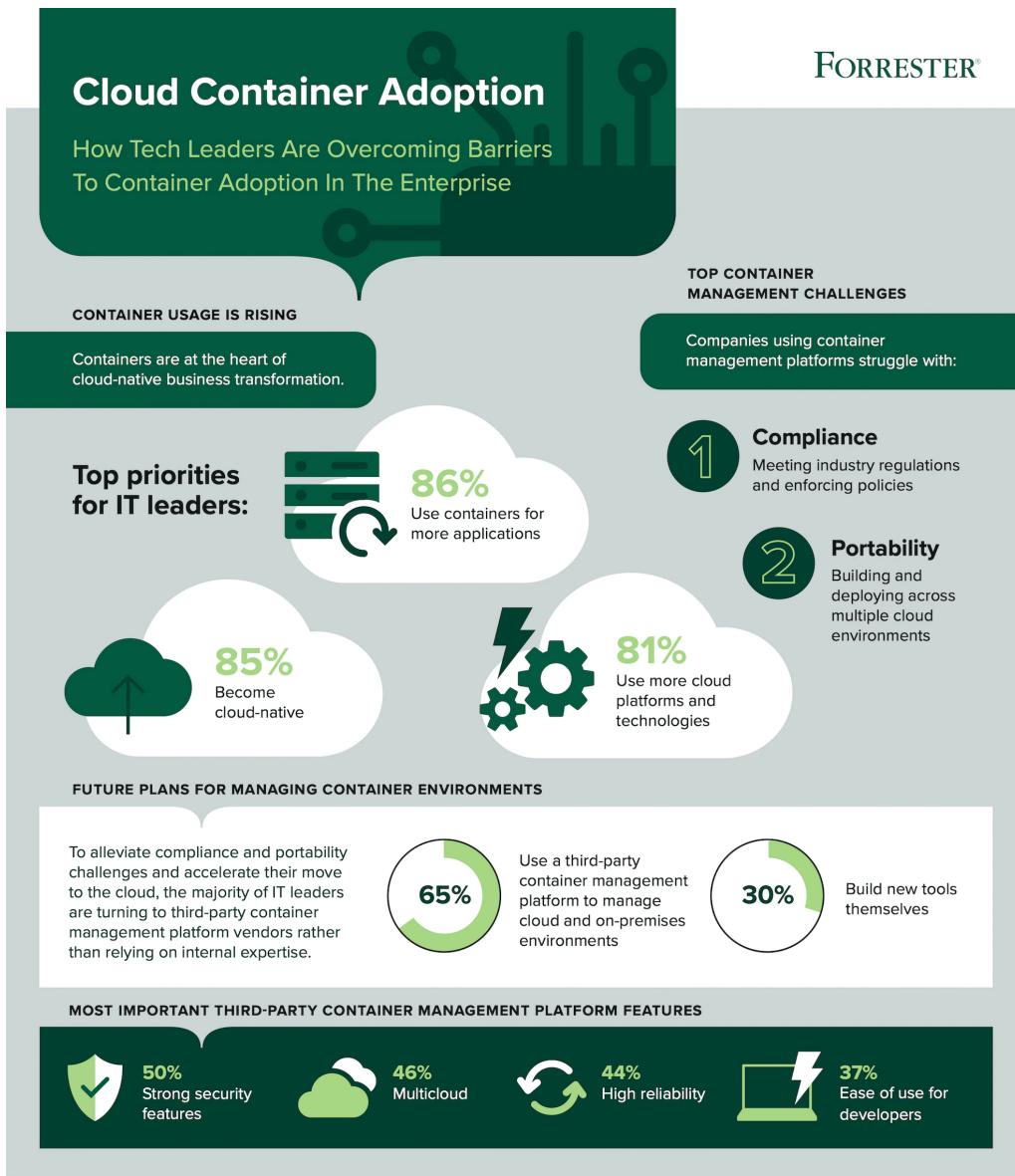


Figura 11: Estadísticas de adopción de tecnologías basadas en contenedores en la nube, 2020 [15].

De entre todos los datos anteriores, es destacable el gran uso de Docker y Kubernetes para gestionar toda esta infraestructura. En 2017, Docker representaba un 99 % de los contenedores en uso. Sin embargo, con la compra de CoreOS por RedHat y el lanzamiento de la OCI ha promovido el nacimiento y establecimiento de nuevas tecnologías de contenedores que le han quitado cuota de mercado a Docker [16]. Actualmente, la distribución queda (figura 12):



Figura 12: Usos de tecnologías de contenedores: Docker domina, seguido por rkt y Mesos [16].

Todo esto nos lleva a ver que si bien aparecen alternativas nuevas Docker sigue siendo la tecnología dominante y la que más adopción está teniendo. Esta competitividad es muy buena ya que permite a Docker y a otras tecnologías de contenedores, como rtk de RedHat (CoreOS), evolucionar, seguir avanzando y mejorando. Lo interesante no es ya usar Docker, rtk o LXC sino que se ha establecido un estándar de contenedores abierto (OCI) y que pone las bases a lo que es una tecnología revolucionaria.

2. Docker

Ahora que ya se han introducido los contenedores, las tecnologías de virtualización y tendencias de uso, se va a explicar cómo funciona Docker en profundidad. Por una parte, se va a ver cómo es la estructura de un contenedor Docker, cómo se comunica con el kernel de Linux, cómo se aísla del resto del sistema y cómo funciona a nivel de discos virtuales, interfaces de red y gestión de recursos.

Por otra parte, se comentarán diversos ejemplos y estructuras básicas que permiten la creación de un contenedor aislado, la comunicación de varios contenedores y el despliegue de una aplicación basada en múltiples contenedores funcionando simultáneamente.

Finalmente, se comentarán tecnologías de orquestación de contenedores, como son los clústers de Kubernetes y Docker Swarm y qué planes hay previstos de cara al desarrollo e innovación de Docker como cliente y gestor de contenedores, para dar pie a un análisis de la seguridad real de los contenedores, en el punto 3.

2.1. Estructura de un Docker

Docker ofrece un mecanismo de comunicación con un entorno aislado llamado contenedor, que ha sido introducido anteriormente. Los contenedores por defecto están aislados, son seguros y empaquetan todo lo necesario para funcionar, por lo que no necesitan nada del sistema que está por debajo.

Internamente, Docker utiliza la arquitectura “cliente–servidor” para gestionar tanto las comunicaciones y contenedores. Por una parte, el cliente de Docker se comunica con el *daemon*,

el cual es el encargado de realizar las tareas pesadas: construir (*build*), levantar (*run*) y distribuir (*distribute*) los contenedores Docker.

Lo interesante del servicio en ejecución de Docker es que puede funcionar o bien en la misma máquina que el cliente o bien sobre otra máquina diferente. Esto es gracias a que la comunicación del cliente con el servicio se realiza mediante *sockets* de UNIX, que proveen de una alta velocidad; o bien mediante interfaces de red, haciendo uso de la API REST de Docker.

La arquitectura Docker

Toda la arquitectura de Docker se puede resumir en la siguiente figura (figura 13):

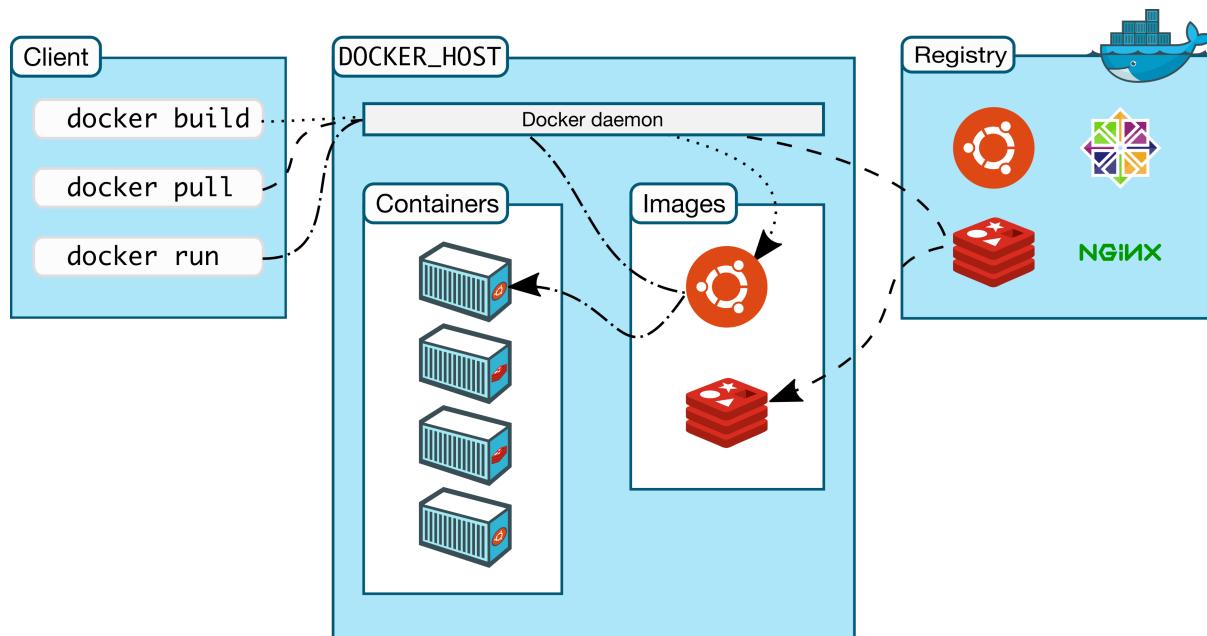


Figura 13: Arquitectura cliente–servidor de Docker [6].

De la imagen anterior destacan tres bloques principales: el cliente, el servidor (*docker host*) y el registro (*registry*). Por una parte, el cliente es la principal forma de comunicarse con el servicio de Docker. Ejecutando comandos como `docker run` se envían al servicio peticiones mediante la API REST interna que gestionan los contenedores.

Por su lado, el servicio está gestionado por el *Docker daemon*, llamado `dockerd`. Dicho servicio escucha de forma activa las peticiones API entrantes y gestiona los objetos de Docker, como las imágenes, contenedores, interfaces de red y volúmenes. Además, un servicio puede comunicarse con otros servicios para gestionar a su vez otros contenedores.

Finalmente, los registros de Docker (no confundir con los *logs*, que también se traduce como “registro”) son un repositorio en donde se almacenan imágenes Docker. Por defecto, se hace uso de Docker Hub, que es el registro principal y al cual el servicio de Docker solicita imágenes cuando no las encuentra, y en donde cualquier usuario puede publicar la imagen que quiera. Pero, además, una persona puede hospedar su propio registro privado (al igual que si se desplegase un servidor Nexus).

Uno de los conceptos fundamentales que se han mencionado anteriormente son los “objetos

Docker". Esa nomenclatura se usa para agrupar y mencionar a todo aquello que se crea y genera cuando se trabaja con el servicio de Docker: imágenes, contenedores, interfaces de red, *plugins*, volúmenes y demás.

Imágenes

Una imagen conforma una plantilla de solo lectura la cual contiene instrucciones para crear un contenedor Docker. Por lo general, las imágenes se basan en otras ya existentes con configuraciones adicionales.

Un caso habitual es una imagen basada en `ubuntu-server` sobre la cual se instala un servidor Apache y la aplicación NodeJS que hemos desarrollado. Con esto, tendríamos una imagen la cual se basa en Ubuntu, que ejecuta un servidor Apache y que tiene una aplicación NodeJS, junto con los ajustes pertinentes para un correcto funcionamiento, definiendo así una imagen nueva (figura 14):

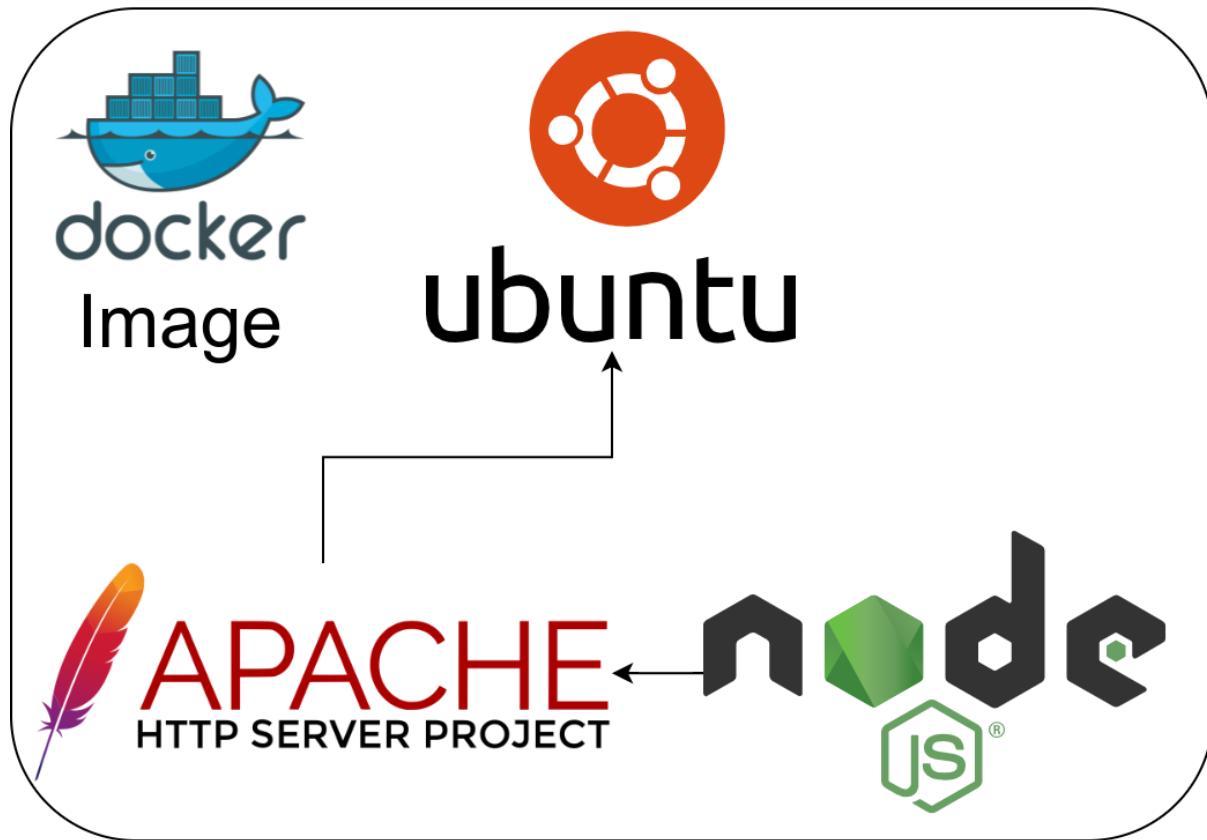


Figura 14: La nueva imagen que habríamos definido basándose en las premisas anteriores.

Cuando se quiere crear una nueva imagen se utiliza un fichero `Dockerfile` el cual incluye las directivas necesarias para construir una imagen desde cero (o basándose en una ya existente). Además, el funcionamiento de este tipo de ficheros es muy similar a los `Makefile` en tanto que, cuando se realiza algún cambio sobre el mismo, solo se reconstruyen en aquellas imágenes que hayan cambiado las capas modificadas. Esto se traduce en imágenes mucho más pequeñas, ligeras y rápidas en comparación, por ejemplo, a las máquinas virtuales.

Contenedores

Un contenedor es una instancia de una imagen que puede ser ejecutada. Las operaciones básicas sobre contenedores son: crear, iniciar, parar, mover o eliminar. Además, se pueden conectar una o varias redes, volúmenes de datos o inclusive definir y crear una nueva imagen a partir del estado actual.

Por defecto, un contenedor está bastante bien aislado del resto de contenedores en la máquina anfitriona. Sin embargo, se puede definir y controlar cómo de aislada está una red, un almacenamiento o los subsistemas que están por debajo.

Es fundamental tener en cuenta que un contenedor está directamente definido por la imagen que lo crea y por las configuraciones iniciales que se le dan en el momento de la creación. Sin embargo, todos los cambios efectuados durante su ciclo de vida que no sean de imagen o de configuración desaparecerán una vez el contenedor se detenga y se elimine (esto incluye todo el sistema de ficheros que hay por debajo).

Almacenamiento

Dada la situación anterior, es necesario buscar alguna manera de persistir la información de los contenedores. Por defecto, los datos que se generan y gestionan en un contenedor de Docker son directamente gestionados por el servicio de Docker y se trabaja con ellos sobre una capa escribible asociada al contenedor [17]. Esto se traduce en:

- Los datos no son persistentes, por lo que eliminar el contenedor eliminará la información.
- La capa escribible de un contenedor está asociada a dicho contenedor, por lo que resulta complejo para otros procesos extraer información de ella.
- Además, dicha capa está directamente asociada a la máquina anfitriona en donde el contenedor está ejecutándose, por lo que es muy complejo mover los datos de un sitio a otro.
- Realizar escrituras sobre la capa escribible de un contenedor necesita de un driver que gestione el sistema de ficheros. Usando el kernel de Linux, este driver ofrece una sistema de ficheros UnionFS, configurado a partir de la unión de varios sistemas de ficheros [18]. Esto conlleva una penalización en rendimiento en comparación con el uso de volúmenes de datos, que trabajan directamente sobre el sistema de ficheros del anfitrión.

Por defecto, existen dos formas de persistir los datos de un contenedor: mediante el uso de volúmenes o mediante *bind mounts*, es decir, asociar un directorio en el host con un directorio en el contenedor.

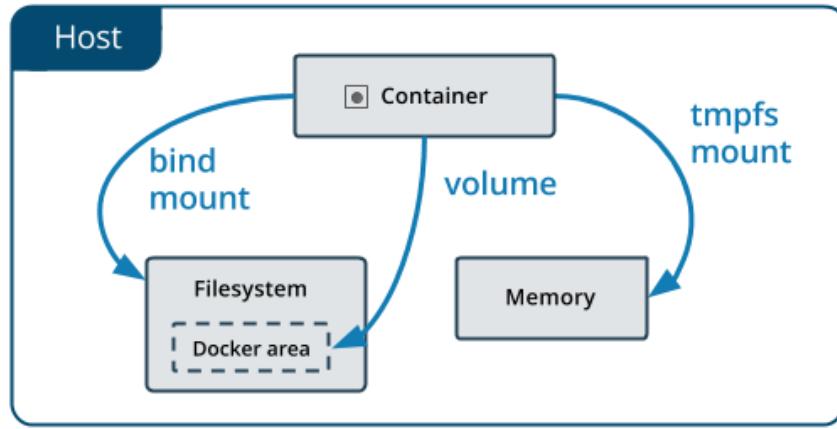


Figura 15: Posibles ubicaciones en donde se almacena la información en contenedores Docker [17].

— Volúmenes

Los volúmenes ofrecen un almacenamiento persistente completamente gestionado por Docker, por lo que hay que crearlos de forma explícita con “`docker volume create`”.

Cuando se crea un volumen, los datos se alojan directamente en la máquina anfitriona. Si se asocia a un contenedor, el volumen monta como un directorio dentro del mismo, mostrando un funcionamiento similar a *bind mounts*. La principal diferencia es que los volúmenes ofrecen un entorno completamente aislado de almacenamiento, gestionado por Docker y portable.

Aprovechando lo anterior, es posible montar un volumen en varios contenedores abriendo la posibilidad de que compartan datos entre ellos. Además, los volúmenes son independientes de los contenedores que los usan: si ningún contenedor usa un volumen, este sigue existiendo hasta que se elimine manualmente (“`docker volume prune`”).

Por otra parte, como los volúmenes son gestionados por Docker permite disponer de *volume drivers* para almacenar datos en equipos remotos.

Para resumir, se indican las características principales de los volúmenes [19]:

- Son fáciles de manejar, hacer copias de seguridad y de migrar a otros servidores.
- Se pueden gestionar completamente desde la interfaz de línea de comandos de Docker.
- Funcionan tanto en Windows como en Linux.
- Están diseñados para que puedan ser compartidos por varios contenedores de forma segura.
- Se pueden almacenar los datos en equipos remotos o proveedores de servicios en la nube.
- El rendimiento en plataformas paganas (MacOS y Windows) es mejor que con *bind mounts*.
- No incrementan el tamaño del contenedor sino del volumen en sí.

Los volúmenes se almacenan en el área de Docker, aislados del sistema dentro del sistema (figura 16):

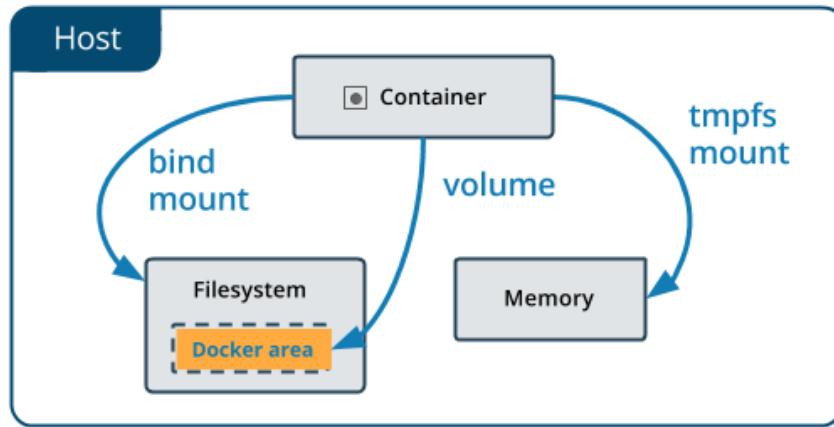


Figura 16: Ubicación del almacenamiento de los volúmenes en Docker [19].

– **Bind mounts**

Los montajes de sistema de ficheros dentro de los contenedores llevan existiendo desde el lanzamiento inicial de Docker. Su funcionamiento es simple: utilizando los mecanismos de los sistemas operativos anfitriones, un directorio (o fichero) en el equipo anfitrión se monta dentro del contenedor en una ruta en específico. Si el fichero o directorio no existe, se crea bajo demanda en el momento de la creación del contenedor.

Es importante tener en cuenta que este tipo de sistema de ficheros está mucho más limitado que un volumen y pueden suponer un gran fallo de seguridad en tanto a que es posible acceder a ficheros sensibles del sistema.

Hay que tener en cuenta que los contenedores Docker siempre se ejecutan como super usuario (administrador), por lo que un proceso malicioso podría editar, modificar, leer y eliminar ficheros fundamentales del sistema anfitrión si no se ha tenido cuidado con el directorio a montar.

Sin embargo, es una opción muy interesante para almacenar datos que son modificados con cierta regularidad o que no interesa “persistirlos”. Por ejemplo, ficheros de configuración (para cambiarla bajo demanda), directorios que contengan logs, etc.

Los *bind mounts* se almacenan en el sistema de ficheros anfitrión (figura 17):

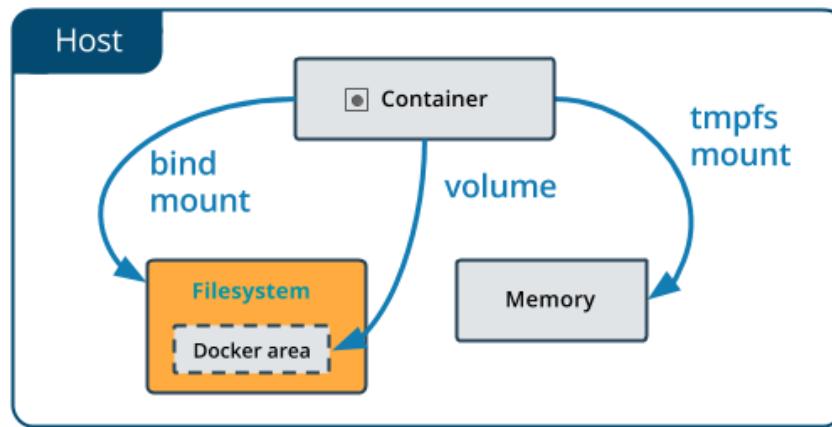


Figura 17: Ubicación del almacenamiento de los *bind mounts* en Docker [20].

– *¿Cuándo usar volúmenes o bind mounts?*

Basándose en la documentación oficial de Docker [17], hay unos casos para unos o para otros, según se requiera (tabla 1):

	Compartir datos entre contenedores	Gestionar ajustes	Estructura del FS	Copias de seguridad	Datos en la nube	Alto rendimiento	Versiones del código fuente
Volúmenes	✓	✗	✓	✓	✓(nativo)	✓(en Docker Desktop)	✗
Bind mounts	✗	✓	✗	✗	✓(usando un sistema de ficheros en la nube, como SAMBA)	✓(depende del sistema de ficheros del anfitrión)	✓

Cuadro 1: Cuándo usar un tipo de almacenamiento persistente u otro, de elaboración propia basándose en la documentación [17].

Interfaces de red

Docker ofrece un potente mecanismo de gestión de redes para permitir la comunicación entre contenedores y con elementos externos. Una de las motivaciones principales detrás de la gestión propia de la red por parte de Docker es la de que las aplicaciones no sepan si están dentro de un contenedor o si tienen que comunicarse con el exterior o con una carga de trabajo externa a Docker, sino que van a comunicarse directamente sin necesidad de ninguna configuración externa.

Indiferentemente de si el servicio se está ejecutando en una máquina Linux y otro en una máquina Windows, la comunicación entre ellas se va a realizar completamente independiente a la plataforma.

La gestión de las redes y aislamiento de las mismas se realiza mediante una manipulación directa de *iptables*. Esto se hace así porque el *firewall* nativo de Linux tiene una grandísima potencia, es muy configurable y se ejecuta directamente a nivel de kernel, lo cual reduce el *overhead* que pudiera presentar si fuese una aplicación a nivel de usuario.

Esto conlleva que las rutas del *firewall* que se hubiesen creado previamente deben aparecer antes de las creadas por Docker (para que tengan mayor peso) y permiten llevar la ejecución de contenedores Docker a distintos tipos de *hardware*: servidores, routers (en donde Docker actúa como router) y equipos embebidos.

Por defecto, Docker expone los siguientes drivers [21]:

- **bridge**: el driver que se usa por defecto, si no se especifica ninguno. Se destina este tipo de driver cuando las aplicaciones se ejecutan de forma aislada y solo necesitan comunicarse con el exterior o, si por el contrario, hace falta que múltiples contenedores se comuniquen entre sí.
- **host**: elimina el aislamiento de red con el sistema anfitrión, usando la interfaz del sistema (por lo que se debe ajustar el *firewall* y la red, si necesita alguna característica especial).
- **overlay**: interconecta múltiples servicios de Docker (que pueden estar en máquinas distintas) para facilitar la comunicación entre clústers de Docker Swarm o contenedores entre sí.
- **macvlan**: asigna una dirección MAC al contenedor de forma que parezca que es una máquina física, por lo que se realiza el enrutamiento según las direcciones MAC.
- **none**: deshabilita todas las comunicaciones de red para el contenedor, se suele usar conjunto a un driver personalizado.
- **Network plugins**: cada persona puede desarrollar su propio driver de red para que cumpla con una función específica.

Interfaz a bajo nivel

Docker, para funcionar, necesita del kernel de Linux por debajo. ¿Por qué esta caracterización? Todo se debe a medidas de seguridad.

Por lo general, un contenedor de Docker se asemeja mucho a los contenedores LXC de Linux nativos, incluyendo entre otros características de seguridad. Si observamos los requisitos de la interfaz Docker (figura 18) vemos que inclusive hace uso de la librería de LXC:

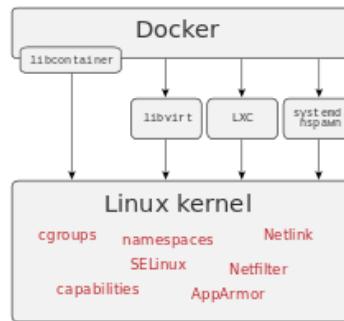


Figura 18: Interfaz de Docker con respecto al kernel de Linux. Fuente: Wikipedia [22].

El principal mecanismo de aislamiento son los *Linux kernel namespaces*, consistentes en una “venda en los ojos” hacia los procesos que se ejecutan dentro de un contenedor.

Lo siguiente que entra en juego son los *control groups* (cgroups), necesarios para limitar el acceso a los recursos del sistema.

Todas estas capas se analizarán en mayor profundidad más adelante, en el punto 3.

2.2. Creación de un contenedor

La creación de un contenedor siempre se realiza de la misma manera: mediante un fichero Dockerfile. Los Dockerfile son ficheros del estilo de los Makefile que contienen unas reglas básicas que definen lo que será una imagen de un contenedor.

El comando que crea una imagen es `docker build`, el cual toma las instrucciones que aparecen en el fichero y las va ejecutando una a una hasta que se produce un error o finaliza correctamente.

Un fichero Dockerfile cuenta con multitud de directivas que permiten personalizar y configurar cómo va a funcionar [23]. Aquí se van a introducir algunas de las más importantes (o las más usadas) [24]:

- **FROM:** define una imagen base de partida sobre la que construir.
- **ADD:** copia ficheros y directorios dentro de la imagen del contenedor. Además, acepta URLs como parámetro y las descarga directamente dentro.
- **RUN:** añade capas a la imagen base, instalando aplicaciones, librerías y componentes.
- **CMD:** especifica qué comandos se deben ejecutar al iniciarse el contenedor. Es importante tener en cuenta que solo puede existir una instrucción CMD en el fichero Dockerfile (si no, se usa solo el último que aparezca). Su funcionalidad principal es la de establecer los valores por defecto de un contenedor en ejecución.

- ENTRYPPOINT: es la instrucción “principal” de un Dockerfile. Especifica el punto de entrada de un contenedor que se quiere que funcione como un ejecutable. Por ejemplo, el siguiente comando “`docker run -it --rm -p 80:80 nginx`” ejecuta un servidor NGINX de forma interactiva, publica el puerto 80 y, cuando finaliza su ejecución, se elimina.
- ENV: define las variables del entorno del contenedor que se usarán en tiempo de ejecución.
- COPY: con una funcionalidad similar a ADD pero con más limitaciones.
- EXPOSE: define en qué puertos la aplicación estará escuchando.
- USER: especifica el UID (o el nombre del usuario) que se usará internamente en el contenedor para ejecutar las aplicaciones.
- VOLUME: define volúmenes de datos a crear o un punto de montaje del contenedor en tiempo de ejecución.
- WORKDIR: especifica la ubicación en donde se ejecutará el comando en tiempo de ejecución.
- LABEL: especifica etiquetas del contenedor en forma de metadatos. Se conforman de parejas clave–valor que especifican información relativa a la imagen dentro del contenedor como, por ejemplo, la versión, el nombre del paquete, etc.

Por lo general, un Dockerfile comienza con la especificación de una imagen de partida (ya que no es habitual crear una imagen desde cero) y, a continuación, se procede a instalar diversos paquetes y capas que puedan ser necesarias para nuestra aplicación. A continuación se copian los paquetes y requisitos de la aplicación dentro de la imagen y se ejecuta la compilación o preparación del paquete, si fuera necesario. Finalmente, se especifica el punto de entrada del contenedor y los argumentos que recibirá, si recibe. Además, si es necesario se añaden los puertos expuestos y volúmenes necesarios para el funcionamiento.

En el código 1 se tiene un ejemplo de un Dockerfile completo [25]:

```
1 # syntax=docker/dockerfile:1
2 FROM golang:1.16-alpine AS build
3
4 # Install tools required for project
5 # Run `docker build --no-cache .` to update dependencies
6 RUN apk add --no-cache git
7 RUN go get github.com/golang/dep/cmd/dep
8
9 # List project dependencies with Gopkg.toml and Gopkg.lock
10 # These layers are only re-built when Gopkg files are updated
11 COPY Gopkg.lock Gopkg.toml /go/src/project/
12 WORKDIR /go/src/project/
13 # Install library dependencies
14 RUN dep ensure -vendor-only
15
16 # Copy the entire project and build it
17 # This layer is rebuilt when a file changes in the project directory
18 COPY . /go/src/project/
19 RUN go build -o /bin/project
20
21 # This results in a single layer image
```

```
22 FROM scratch
23 COPY --from=build /bin/project /bin/project
24 ENTRYPOINT [ "/bin/project" ]
25 CMD [ "--help" ]
```

Listing 1: Ejemplo de Dockerfile para una aplicación Go [25].

En el fichero anterior se establece una imagen de partida `golang:1.16-alpine` que contiene los binarios de Go en una distribución de muy poco peso (proyecto Linux Alpine). Esta imagen se obtiene desde Docker Hub.

A continuación, instala el paquete `git` y el gestor de dependencias de Go. Después, copia el proyecto en sí e instala las dependencias en la ruta `/go/src/project`. Finalmente, copia el directorio actual en dicha ruta y define una nueva imagen partiendo de la de Golang en donde se ejecuta el proyecto compilado con las opciones `--help` por defecto.

Cuando se define un contenedor es recomendable instalar solo aquellas dependencias que sean necesarias. Esto permite una mayor y mejor mantenibilidad, reduce la complejidad y el tiempo de construcción de la imagen.

Por otra parte, se recomienda encarecidamente desacoplar la aplicación: por ejemplo, un servidor web posiblemente requiera de varias aplicaciones en ejecución. Es recomendable separarlas en contenedores y habilitar la comunicación entre ellos a encapsularlo todo en un único contenedor. Esto facilita, entre otros, un escalado horizontal en donde si se requieren de más imágenes se crean.

Además, el número de capas se debe mantener lo más pequeño posible: las instrucciones `RUN`, `COPY` y `ADD` crean capas, mientras que otras instrucciones solo crean imágenes intermedias.

Finalmente, es importante construir el Dockerfile teniendo en cuenta que Docker usa una caché interna. Si se quiere deshabilitar se puede hacer con la opción `--no-cache=true` en el comando `docker build`. Sin embargo, es recomendable hacer uso de la caché interna de Docker ya que agiliza el proceso de construcción.

2.3. Comunicación entre contenedores

La comunicación entre contenedores presenta dos formas posibles:

1. O bien mediante la comunicación mediante una red o un *socket*.
2. O bien mediante ficheros compartidos en un volúmen o carpeta.

Evidentemente, cada una presenta sus ventajas. Sin embargo, la primera opción es la escogida por lo general. ¿Por qué?

Habilitar la comunicación mediante una red asegura que solo los contenedores especificados reciben la información en cuestión. Si se usa un directorio compartido se corre el riesgo de que otra aplicación pueda ver la información y eso no es seguro.

Por otra parte, las comunicaciones vía red facilitan la escucha pasiva mediante el uso de puertos: si se tiene un puerto publicado, basta con escuchar allí a la espera de peticiones. Si

se pretende realizar una comunicación mediante un directorio compartido es necesario o bien hacer *polling* cada ‘x’ segundos o bien usar algún mecanismo del kernel (como *inotify*) para ser notificados cuando se realice alguna acción sobre un fichero en cuestión.

Finalmente, comunicaciones basadas en la red permiten identificar quién es el emisor que está al otro lado, escuchando o recibiendo información. Sin embargo, en comunicaciones basadas en ficheros cualquiera puede acceder y no se sabe necesariamente qué proceso es el último que ha editado o leído un fichero.

La forma de comunicar dos contenedores vía red se realiza principalmente mediante puentes (conexiones tipo *bridge*). Este tipo de interfaz es nativa de Docker y muy fácil de configurar. Por defecto, un contenedor usará este tipo de red para realizar sus comunicaciones con el exterior y con otros equipos, por lo que usarla para conectar varios contenedores es sencillo. Existen dos formas de hacerlo:

1. Durante la creación, especificando la red a la que conectarse.
2. En tiempo de ejecución, usando el cliente Docker para indicarle a un contenedor que se debe unir a una red.

Conexión

Para realizar la conexión entre contenedores se utilizan siempre redes virtuales. El tipo de red puede variar según se necesite, pero principalmente se usan las tipo puente (mencionadas anteriormente) y las redes *overlay*, más específicas para casos más concretos que se mencionarán más adelante.

En estas redes el contenedor cuenta con una dirección IP única asignada por el servicio Docker (*dockerd*) y, opcionalmente, un *hostname* que permite la identificación directa y sencilla del contenedor.

Por defecto, Docker crea una interfaz *bridge* para el contenedor y se puede usar para comunicar dos contenedores. Sin embargo, el nombre de las redes no es fácilmente accesible y menos el de los contenedores. La idea principal radica en el uso de la IP única asignada a dicha interfaz: si se obtiene la dirección de un contenedor, otro podrá enviar información directamente a esa IP.

La forma de configurar esta conexión es la siguiente:

1. Se comprueba que la red a la que se quieran conectar los contenedores esté en ejecución. Esto se consigue gracias al comando `docker network ls`:

javinator9889@Tony-MkIII ~ docker network ls			
NETWORK ID	NAME	DRIVER	SCOPE
f8f93c7fb28e	bridge	bridge	local
e4be40945267	cachet-docker_default	bridge	local
b86463867cae	cp-all-in-one_default	bridge	local
4d0ebd5b00c8	docker_gwbridge	bridge	local
9926f93b44dc	host	host	local
nb0bvgbg8lts	ingress	overlay	swarm
9a068883a020	none	null	local

2. Se inician los contenedores (o se obtiene su ID) para posteriormente, poder obtener su IP. Se puede obtener el identificador de un contenedor mediante el comando `docker container ls`:

```
javinator9889@Tony-MkIII ~ docker container ls
CONTAINER ID IMAGE COMMAND
254299cf59d9 postgres:12-alpine "docker-entrypoint.s..."
fa31449d87af portainer/portainer-ce "/portainer"
789dc8b56c8 redis:alpine "docker-entrypoint.s..."
```

3. Finalmente, se obtiene la dirección IP interna del contenedor. Para ello, se pueden usar herramientas externas como Portainer o directamente desde el terminal mediante comandos: `docker inspect <container-id> | grep IPAddress`:

```
javinator9889@Tony-MkIII ~ docker inspect 254 | grep IPAddress
"SecondaryIPAddresses": null,
"IPAddress": "",
"IPAddress": "172.20.0.2",
```

Con la dirección IP ya podemos empezar a comunicarnos con el contenedor, tanto desde dentro de Docker como desde la máquina anfitriona. Esto se puede comprobar fácilmente haciendo un ping a la IP (figura 19):

```
X javinator9889@Tony-MkIII ~ ping -c 5 172.20.0.2
PING 172.20.0.2 (172.20.0.2) 56(84) bytes of data.
64 bytes from 172.20.0.2: icmp_seq=1 ttl=64 time=0.036 ms
64 bytes from 172.20.0.2: icmp_seq=2 ttl=64 time=0.094 ms
64 bytes from 172.20.0.2: icmp_seq=3 ttl=64 time=0.109 ms
64 bytes from 172.20.0.2: icmp_seq=4 ttl=64 time=0.090 ms
64 bytes from 172.20.0.2: icmp_seq=5 ttl=64 time=0.092 ms

--- 172.20.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4098ms
rtt min/avg/max/mdev = 0.036/0.084/0.109/0.025 ms
```

Figura 19: Desde la máquina anfitriona nos podemos comunicar con el contenedor mediante la IP.

La opción anterior sin embargo es bastante tediosa y no permite realizarlo fácilmente en pocas líneas. Es por ello por lo que existe la opción de redes definidas por el usuario (*user-defined bridge network* [26]).

Cuando un grupo de contenedores se unen a una red tipo puente definida por el usuario ya no es necesario tener el control sobre la dirección IP de cada contenedor sino que basta con referenciar al contenedor por su nombre (figura 20):

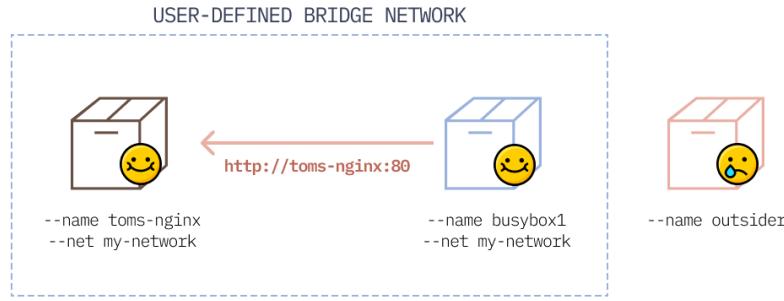


Figura 20: Interfaz de red tipo puente definida por el usuario para facilitar la comunicación entre contenedores [26].

De esta forma, si se ejecuta una base de datos dentro de un contenedor (por ejemplo, una base tipo PostgreSQL), la conexión se puede realizar directamente escribiendo:

```
postgresql://pgsql-container:5432
```

lo cual facilita mucho la labor de depuración y desarrollo.

El proceso de creación y conexión en este caso es de la siguiente forma:

1. Se crea una red definida por el usuario de tipo puente, mediante el comando `docker network create <name>`. Por debajo Docker se encargará de gestionar todo lo necesario:

```
javinator9889@Tony-MkIII ~ docker network create ssr
f3c6a2bcc7870ba86a4c135105c6fedb68bf2b0366c5054749afcd26e7916374
javinator9889@Tony-MkIII ~ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
f8f93c7fb28e   bridge    bridge    local
e4be40945267   cachet-docker_default  bridge    local
b86463867cae   cp-all-in-one_default  bridge    local
4d0ebd5b00c8   docker_gwbridge    bridge    local
9926f93b44dc   host      host      local
nb0vqbq8lts   ingress   overlay  swarm
9a068883a020   none     null     local
f3c6a2bcc787   ssr      bridge    local
```

2. A continuación se crea el contenedor y se le asigna la red que acabamos de crear. Es importante en este paso asignarle también un nombre al contenedor para que sea fácilmente accesible. Lo primero se consigue con la opción `--network <network-name>` y lo segundo con la directiva `--name <container-name>`:

```
javinator9889@Tony-MkIII ~ docker run --rm --network ssr --name=nginx -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
69692152171a: Pull complete
30afc0b18f67: Pull complete
596b1d696923: Pull complete
febe5bd23e98: Pull complete
8283eee92e2f: Pull complete
351ad75a6cfa: Pull complete
Digest: sha256:6d75c99af15565a301e48297fa2d121e15d80ad526f8369c526324f0f7ccb750
Status: Downloaded newer image for nginx:latest
9b6613c2c8c40397f528f6599cf5f68d9439851a9bd891d96978592f687be2b1
```

3. Finalmente, creamos el/los otro(s) contenedor(es) y les asociamos igualmente la red que hemos creado. En este caso, vamos a ejecutar una terminal básica que va a realizar una petición al servidor de NGINX que acabamos de levantar con el comando wget. Para acceder se pone directamente el nombre del contenedor de NGINX y el puerto al que se quiere acceder, no hace falta usar la IP:

```
javinator9889@Tony-MkIII ~ docker run --rm --network ssr -it busybox sh
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
b71f96345d44: Pull complete
Digest: sha256:930490f97e5b921535c153e0e7110d251134cc4b72bbb8133c6a5065cc68580d
Status: Downloaded newer image for busybox:latest
/ # wget -q -O- nginx:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Lo interesante de este método no es solo que no haga falta saber la IP del contenedor (la cual entre reinicios puede cambiar) sino que además permite la conexión “en caliente” de un contenedor a la red: si durante la creación no le hemos asignado la red no habría problema ya que existe un método de poder unir el contenedor a la red. Este comando es `docker network → connect <network-name> <container-name/ID>` y permite gestionar las interfaces de red de un contenedor fácilmente [27]. Es el reemplazo del antiguo comando `--link`, ya en desuso [28] el cual unía dos contenedores por medio de una red puente.

Redes *overlay*

Otra forma de conectar contenedores es mediante las redes *overlay*. Dichas redes permiten conectar múltiples clústers de Docker (en particular, *Swarm*) entre sí para facilitar el intercambio de información. Sin embargo, el modo de funcionamiento ya no se limita únicamente a una red local sino a una red WAN para la que es necesario añadir orquestación mediante Kubernetes o Docker Swarm.

El procedimiento sin embargo es muy similar al de crear una red tipo puente personalizada, solo se alteran algunos pasos:

1. Inicializamos un clúster de *Swarm* y exponemos la IP a la cual otros nodos podrán unirse. Esto se hace con:

```
docker swarm init --advertise-addr <IP-address>
```

2. Tras la creación se generará un token el cual se usará para unirse al clúster de Swarm:

```
docker swarm join --token <Swarm-token>
```

Este paso es necesario ya que, en otro caso, estaremos trabajando sobre el servicio local de Docker y no sobre el clúster de Swarm.

3. A continuación, creamos la red *overlay*. En este caso, el comando difiere en que hay que añadir el tipo de red que se quiere crear:

```
docker network create -d overlay <network-name>
```

```
javinator9889@Tony-MkIII ~ % docker network create -d overlay ssr-overlay
javinator9889@Tony-MkIII ~ % docker network ls
NETWORK ID      NAME        DRIVER    SCOPE
f8f93c7fb28e   bridge      bridge    local
e4be40945267   cachet-docker_default  bridge    local
b86463867cae   cp-all-in-one_default  bridge    local
4d0ebd5b00c8   docker_gwbridge     bridge    local
9926f93b44dc   host        host      local
nb0bvgbg8lts   ingress     overlay   swarm
9a068883a020   none        null      local
f3c6a2bcc787   ssr         bridge    local
ciei355btlh0   ssr-overlay    overlay   swarm
```

- Como se está trabajando sobre Docker Swarm, se han de crear servicios (no contenedores) que se ejecutarán de forma distribuida a lo largo del clúster. En este caso, a modo de demostración se va a crear un servicio de NGINX que sea similar al del ejemplo anterior:

```
docker service create --name nginx-server -d --network ssr-overlay
                           ↪ nginx
```

```
javinator9889@Tony-MkIII ~ % docker service create --name=nginx-server -d --network ssr-overlay nginx
javinator9889@Tony-MkIII ~ % docker service ls
ID          NAME      MODE      REPLICAS  IMAGE
x5nkeccy2653 nginx-server replicated 1/1      nginx:latest
```

- Finalmente, se crea otro servicio que en este caso va a acceder directamente al servicio de NGINX mediante el comando wget:

```
docker service create --network ssr-overlay --name bb -d busybox
                           ↪ wget -q -O- nginx-server:80
```

```
javinator9889@Tony-MkIII ~ % docker service create --network=ssr-overlay --name bb -d busybox wget -q -O- nginx-server:80
javinator9889@Tony-MkIII ~ % docker service logs bb
zsh: correct 'logs' to 'log' [ynae]? n
bb.1.d008v3f9t12j@Tony-MkIII | <!DOCTYPE html>
bb.1.d008v3f9t12j@Tony-MkIII | <html>
bb.1.d008v3f9t12j@Tony-MkIII | <head>
bb.1.d008v3f9t12j@Tony-MkIII | <title>Welcome to nginx!</title>
```

Como se puede ver, también se pueden conectar los contenedores a través de redes *overlay* y usando directamente el nombre del contenedor. Su configuración es bastante más compleja y añade más pasos, que se explicarán con más detalle en el punto 2.5.

2.4. Despliegue de aplicaciones multi-contenedores. docker-compose

Uno de los principales puntos que se han visto durante el desarrollo del documento es la necesidad de comunicar contenedores y de distribuir aplicaciones en varias imágenes Docker.

Un *stack* típico de aplicaciones sería un servidor XAMPP: un contenedor ejecutando Apache que se comunica con una base de datos MySQL y que utiliza un *backend* basado en PHP. Se puede construir una única imagen la cual se componga de esas tres aplicaciones y sus dependencias. Sin embargo, según se recomienda desde Docker, lo ideal para no añadir demasiada

complejidad sería crear un contenedor mínimo para cada aplicación que funcione de forma aislada del resto pero que se comunique con ello.

Si bien esta tarea se podría realizar manualmente con los comandos que se han visto anteriormente, existe una herramienta llamada “`docker-compose`” la cual gestiona clústers de contenedores que se ejecutan a la vez y que dependen unos de otros.

¿Qué es Docker Compose?

Docker Compose es una herramienta basada en Python que se usa para definir aplicaciones multicontenedor dentro de Docker. Se basa en el formato de ficheros YAML para definir los servicios de la aplicación y, con un único comando, preparar su ejecución y trabajar con dichos servicios [29].

La definición de un servicio de Compose se define en tres pasos:

1. Definir los ficheros `Dockerfile` de cada uno de los entornos de la aplicación.
2. Definir qué servicios componen la aplicación en un fichero `docker-compose.yml`.
3. Ejecutar el comando “`docker compose up`” para lanzar la aplicación al completo.

Un fichero `docker-compose.yml` tiene una forma como esta (código 2):

```

1 version: "3.9" # optional since v1.27.0
2 services:
3   web:
4     build: .
5     ports:
6       - "5000:5000"
7     volumes:
8       - .:/code
9       - logvolume01:/var/log
10    links:
11      - redis
12    redis:
13      image: redis
14    volumes:
15      logvolume01: {}

```

Listing 2: Estructura típica de un fichero de Docker Compose [29].

La estructura que se sigue habitualmente se define por las siguientes claves en el fichero YAML [30]:

- `version: "XY"` define qué versión de Docker Compose se está usando. Esta línea, aunque opcional desde la versión v1.27.0, es fundamental ya que las características de Docker Compose varían según la versión que se esté usando. Es importante que en este caso una versión posterior no implica que “sea mejor” sino que algunas características cambian de nombre, otras aparecen y otras se eliminan.

- **services** – esta sección define los distintos contenedores que se van a crear. En el ejemplo del código 2, se crean dos servicios: uno primero que es `web` que se construye sobre el directorio actual; y un segundo con nombre `redis` que usa la imagen de Redis desde Docker Hub.
- **build** especifica la ubicación del fichero `Dockerfile`. En el ejemplo del código 2 se usa el directorio actual ‘`.`’ como ubicación para construir el contenedor.
- **ports** especifica los puertos mapeados del contenedor en cuestión. Es el equivalente a la opción `-p` del comando `docker`.
- **volumes** define los volúmenes que se van a usar en el contenedor. En particular es como usar la opción `-v` a la hora de crear un contenedor. Por ende, se puede usar un directorio para hacer *bind mounts*, usar el nombre de un volumen, etc.
- **links** permite unir un contenedor a otro. En este caso, se especifica a qué contenedor se puede acceder.
- **image** permite definir un servicio en un `docker-compose` usando una imagen ya existente en Docker Hub, si no se dispone del `Dockerfile` apropiado.
- **environment** define las variables del entorno del contenedor y sus respectivos valores. Equivalente a la opción `-e` del comando `docker`.

Así, con la estructura del fichero ya definida, usando el comando `docker-compose` se puede [29]:

- Iniciar, parar y reconstruir servicios.
- Ver el estado de los servicios en ejecución.
- Volcar los registros de ejecución en tiempo real (*stream view*).
- Ejecutar un único comando en un servicio.

Tras esta idea se ha construido una forma muy potente y simple de crear aplicaciones multicontenedor. Aprovechando las características de Docker, Compose permite ejecutar distintos entornos aislados en el mismo anfitrión. Por ejemplo, de esta forma, una aplicación Docker Compose puede contar con varios entornos según en donde se quiera ejecutar. Por ejemplo, se puede tener un entorno de desarrollo, otro de CI y uno de producción en el mismo *host*. Esto se consigue gracias a los nombres de proyecto, característica fundamental de Docker Compose para distinguir y aislar entornos.

Otro punto interesante es que se puede actualizar el fichero “`docker-compose.yml`” cuando se necesite y la construcción de los contenedores se realizará únicamente para aquellos que hayan cambiado. Esto da una gran agilidad a la hora de presentar actualizaciones en un entorno de producción y potencia la idea de microservicios, en donde solo se actualiza lo que se necesita.

Esto afecta también a cómo se gestionan los datos de los contenedores. Si el volumen asociado a un contenedor ya existía se sigue usando el mismo, en ningún momento se “tira abajo” y se empieza desde cero.

Finalmente, una de las características más interesantes de Docker Compose es que soporta las variables del entorno. Esto se traduce en que un mismo fichero YAML de Docker Compose puede producir distintas configuraciones según el entorno en que se encuentre. Aprovechando esta característica, se pueden definir y usar ficheros .env para establecer las variables del entorno que se usarán a la hora de desplegar los servicios y trabajar con los contenedores.

Comandos Docker Compose

De entre todas las opciones que ofrece Compose, algunas son muy interesantes por las potencias que ofrecen y sus utilidades [30]:

- `docker-compose build` prepara las imágenes construyendo los servicios, listos para ser lanzados. Si una imagen es del repositorio *online* no se hace nada.
- `docker-compose images` lista las imágenes que se han construido desde el fichero actual.
- `docker-compose stop` detiene los servicios en ejecución actuales.
- `docker-compose run <service>` crea los contenedores para el servicio especificado.
- `docker-compose up` inicia todo el proceso de despliegue de los servicios: primero, construye las imágenes necesarias y luego inicia cada uno de los contenedores especificados.
- `docker-compose ps` lista todos los contenedores del `docker-compose.yml` actual.
- `docker-compose down` detiene todos los contenedores y limpia sus datos, redes e imágenes.

Caso real

A modo de demostración, se va a crear un *stack* Wordpress, NGINX, PHP-FPM y MySQL para desplegar un servidor web Wordpress de forma sencilla (basándose en el ejemplo de Digital Ocean [31]).

Asumiendo que los ficheros de configuración de NGINX ya están preparados así como las dependencias necesarias, se pasa a definir los ficheros de Docker Compose. Por una parte, para la base de datos, se van a definir las credenciales en un fichero .env:

```
1 MYSQL_ROOT_PASSWORD="password-random"
2 MYSQL_USER="wordpress_user"
3 MYSQL_PASSWORD="wordpress-random-password"
```

A continuación, se pasa a definir cada uno de los servicios que compondrá la aplicación. El primero de ellos será la base de datos MySQL (código 3):

```
1 version: '3'
2
3 services:
4   db:
5     image: mysql:8.0
6     container_name: db
7     restart: unless-stopped
```

```

8 env_file: .env
9 environment:
10   - MYSQL_DATABASE=wordpress
11 volumes:
12   - dbdata:/var/lib/mysql
13 command: '--default-authentication-plugin=mysql_native_password'
14 networks:
15   - app-network

```

Listing 3: Servicio de MySQL para el *stack* Wordpress.

Lo primero que se especifica es la versión de Docker Compose que se necesita, en este caso, la 3. Para este caso en particular, se va a usar la imagen de MySQL versión 8.0 (clave `image`) con nombre “db” y que se debe reiniciar hasta que se detenga (clave `restart`). En lo referente a las variables del entorno, se define el nombre de la base de datos asignando la clave `MYSQL_DATABASE` y se define un volumen “`dbdata`” el cual almacenará los datos de MySQL de forma persistente. Finalmente, se especifican algunas opciones iniciales al comando inicial de MySQL y se especifica la red en la que estará conectada.

A continuación, se define el servicio de Wordpress (código 4):

```

1 wordpress:
2   depends_on:
3     - db
4   image: wordpress:5.1.1-fpm-alpine
5   container_name: wordpress
6   restart: unless-stopped
7   env_file: .env
8   environment:
9     - WORDPRESS_DB_HOST=db:3306
10    - WORDPRESS_DB_USER=$MYSQL_USER
11    - WORDPRESS_DB_PASSWORD=$MYSQL_PASSWORD
12    - WORDPRESS_DB_NAME=wordpress
13   volumes:
14     - wordpress:/var/www/html
15   networks:
16     - app-network

```

Listing 4: Servicio de Wordpress.

En este caso, la configuración es similar al de MySQL pero se añade una directiva que define el orden de inicio de los contenedores (`depends_on`). Se va a usar la imagen de `wordpress:5.1.1-fpm` que es la versión de Wordpress 5.1.1 con PHP-FPM por detrás en la versión *alpine*, derivada del proyecto Alpine Linux que busca un tamaño reducido de las imágenes. A continuación, se especifica el fichero que contiene las variables del entorno (`env_file`) y se definen las opciones de Wordpress. Finalmente, se crea un volúmen que contendrá los datos de Wordpress y se asocia al directorio `/var/www/html` dentro del contenedor y se define la red a la que se conecta.

Por último, pero no menos importante, se define el servidor web en sí. En este caso, se usa NGINX (código 5):

```

1 webserver:
2   depends_on:
3     - wordpress
4   image: nginx:1.15.12-alpine

```

```

5  container_name: webserver
6  restart: unless-stopped
7  ports:
8    - "80:80"
9  volumes:
10   - wordpress:/var/www/html
11   - ./nginx-conf:/etc/nginx/conf.d
12 networks:
13   - app-network

```

Listing 5: Servicio de NGINX.

De la configuración anterior es importante fijarse en que, por una parte, se expone el puerto 80 (HTTP) al exterior (opción `ports`) y se reutiliza el volúmen de Wordpress. Además, se hace un *bind mounts* del fichero “`nginx-conf`” dentro del directorio `/etc/nginx/conf.d`. Esto permite realizar cambios en la configuración de NGINX y que se puedan aplicar directamente.

Finalmente, al final del fichero, se añade la especificación de la red y de los volúmenes (código 6):

```

1 volumes:
2   wordpress:
3   dbdata:
4
5 networks:
6   app-network:
7     driver: bridge

```

Listing 6: Especificación de los volúmenes y de las redes.

Quedando un fichero `docker-compose.yml` así:

```

1 version: '3'
2
3 services:
4   db:
5     image: mysql:8.0
6     container_name: db
7     restart: unless-stopped
8     env_file: .env
9     environment:
10      - MYSQL_DATABASE=wordpress
11     volumes:
12       - dbdata:/var/lib/mysql
13     command: '--default-authentication-plugin=mysql_native_password'
14     networks:
15       - app-network
16
17   wordpress:
18     depends_on:
19       - db
20     image: wordpress:5.1.1-fpm-alpine
21     container_name: wordpress
22     restart: unless-stopped
23     env_file: .env
24     environment:
25       - WORDPRESS_DB_HOST=db:3306
26       - WORDPRESS_DB_USER=$MYSQL_USER

```

```
27      - WORDPRESS_DB_PASSWORD=$MYSQL_PASSWORD
28      - WORDPRESS_DB_NAME=wordpress
29  volumes:
30      - wordpress:/var/www/html
31  networks:
32      - app-network
33
34 webserver:
35  depends_on:
36      - wordpress
37  image: nginx:1.15.12-alpine
38  container_name: webserver
39  restart: unless-stopped
40  ports:
41      - "80:80"
42  volumes:
43      - wordpress:/var/www/html
44      - ./nginx-conf:/etc/nginx/conf.d
45  networks:
46      - app-network
47
48 volumes:
49  wordpress:
50  dbdata:
51
52 networks:
53  app-network:
54      driver: bridge
```

Después de realizar varias configuraciones iniciales, el servidor Wordpress ya estará completamente funcional y disponible (ver el ejemplo completo en la web de Digital Ocean [31]).

2.5. “Orquestación” de contenedores

Todo lo anterior lleva al punto crítico en donde se encuentran los contenedores y tecnologías como Docker actualmente: ¿cómo se automatiza el despliegue? ¿Cómo se realizan escalados automáticos? ¿Cómo se guardan recursos, se conservan y se reutilizan más tarde? Bienvenidos a la orquestación de contenedores.

Actualmente las herramientas de orquestación están en boga y son la clave del desarrollo basado en contenedores. Ofrecen una gran polivalencia y flexibilidad a la hora de desplegar aplicaciones a lo largo de clústers distribuidos de una forma relativamente sencilla y accesible.

Herramientas como Kubernetes, Minikube, AWS EKS o Docker Swarm son de las más sonadas y utilizadas actualmente. El primero de ellos es actualmente el más usado a nivel mundial. Kubernetes es una herramienta de código abierto diseñada por Google que permite orquestar contenedores a lo largo de una red distribuida de nodos. Minikube por su parte es una versión local de Kubernetes con las mismas características, muy útil para desplegar aplicaciones en entornos locales y hacer pruebas antes de un despliegue global. AWS EKS es la solución nativa de Amazon para su servicio en la nube de orquestación. Se basa en Kubernetes y permite realizar una gestión individualizada de los contenedores para ajustar el pago y los recursos destinados. Finalmente, Docker Swarm es la solución nativa de Docker para la orquestación de contenedores. Se gestiona con el propio CLI de Docker y, aunque no

es tan popular como otras herramientas, está ganando fuerza debido a su gran facilidad de uso.

¿Por qué es importante la orquestación de contenedores?

Es fundamental intentar vislumbrar el porqué han ganado tanto peso estas herramientas de orquestación ya que es la clave de muchas empresas. La organización en contenedores automatiza la implementación, la gestión, la escalabilidad y la conexión en red de los contenedores. Empresas que necesiten implementar y gestionar cientos o miles de *hosts* usando contenedores Linux se benefician directamente de la orquestación de contenedores [32].

En particular, estas tecnologías han visto un gran auge con la aparición de los microservicios organizados en contenedores. Con ellas se pueden desplegar más instancias de un microservicio si la carga del sistema es elevada o reducir los contenedores según sea la demanda. Por otra parte, los microservicios son muy útiles en la gestión de los ciclos de vida de los contenedores, en particular para los equipos de DevOps: una integración correcta en los flujos de trabajo de CI/CD permiten definir la base de las aplicaciones nativas en la nube.

Es por ello por lo que las tecnologías de orquestación han proliferado y permiten definir automatizar y gestionar muchas tareas [32]:

- Preparación e implementación.
- Configuración y programación.
- Asignación de recursos.
- Disponibilidad de contenedores.
- Ajusta o eliminación de contenedores según la carga del sistema.
- Equilibrio de cargas y enrutamiento de tráfico.
- Supervisión del estado de los contenedores.
- Configuración de aplicaciones en función del contenedor en que se vayan a ejecutar.
- Protección de las interacciones entre contenedores.

Con todas estas tareas automatizadas, es más sencillo gestionar aplicaciones distribuidas. Todo esto es necesario debido a la naturaleza ligera y efímera de los contenedores, en donde el ciclo de vida de alguno de ellos puede llegar a ser de unos pocos segundos (como en microservicios). Así, la orquestación viene a ofrecer los siguientes beneficios [33]:

- Simplicidad en las operaciones, el gran beneficio de la orquestación: según lo expuesto anteriormente, las aplicaciones basadas en contenedores pueden introducir una gran complejidad y generar muchísimos datos. Con la orquestación es más simple gestionarlo.
- Resiliencia, permitiendo el reinicio y escalado de los contenedores necesarios según se necesite de forma automática.
- Seguridad, ya que se reduce el error humano.

2.5.1. Herramientas de orquestación

Kubernetes



Figura 21: Logo de Kubernetes [34].

Kubernetes es actualmente la herramienta de orquestación más utilizada por las compañías que basan sus aplicaciones en contenedores [14]. Fue una herramienta de código libre creada por Google y donada a la *Cloud Native Computing Foundation*, quien lo gestiona ahora.

Dado que los contenedores son una buena manera de desplegar aplicaciones, la gestión en entornos de producción se cede a herramientas como K8s que se aseguran de que no exista tiempo de *downtime* (K8s son las siglas de “Kubernetes”, donde el 8 representa las ocho letras entre la ‘K’ y la ‘s’). Por ejemplo, si un contenedor se cae y no ofrece servicio, automáticamente otro se crea y se levanta y suple al anterior [35].

K8s ofrece un *framework* para poder ejecutar sistemas distribuidos de forma resiliente. Gestiona automáticamente el escalado y el control de fallos de la aplicación, ofrece patrones de despliegue y más opciones, entre las que se destacan [35]:

- Descubrimiento de servicios y balanceo de carga: mediante el uso de direcciones DNS o direcciones IP, K8s puede exponer un contenedor a la red. Si por casual el tráfico es elevado, aprovechando el mecanismo anterior se puede redirigir automáticamente a un contenedor con poca carga o que pueda servir la operación de una forma más eficaz.
- Orquestación del almacenamiento: Kubernetes permite gestionar el almacenamiento local, en la nube o en proveedores para los contenedores que lo necesiten.
- *rollouts* y *rollbacks*: mediante la descripción de los estados en los que se pueden encontrar los contenedores (como el estado “deseable”), Kubernetes puede modificar el estado para alcanzar dicho estado deseable o bien volver a un estado controlable. Un ejemplo sería el despliegue de nuevos contenedores, eliminar los ya existentes y adoptar sus recursos para continuar desde el estado en que se dejó.
- Gestión de los recursos: se puede configurar un clúster de Kubernetes para que use una cantidad específica de CPU y de RAM y automáticamente se distribuirán los servicios para hacer el uso más eficiente posible de los recursos.
- Autocuidado: si un contenedor falla, se reinicia; si no funciona como se espera, se reemplaza; si no hay respuesta en un plazo de tiempo, se finaliza. Todo esto se hace de forma transparente al usuario y nunca se ofrecen servicios que todavía no estén disponibles.

- Gestión de “secretos” y configuraciones: K8s facilita la gestión de información sensible como contraseñas, claves OAuth, etc., permitiendo la actualización de dichos valores sin necesidad de desplegar de nuevo la aplicación y sin exponer dicha información al exterior.

Sin embargo, es fundamental tener en cuenta qué operaciones nunca va a realizar Kubernetes y que lo diferencian de otras soluciones o alternativas. Por una parte, Kubernetes no es una PaaS (*Platform as a Service*) tradicional. Esto se debe a que K8s no es monolítico y aunque comparte muchas características comunes con un PaaS otras son libres, configurables y se ajustan por el usuario.

Por otra parte, las aplicaciones no están limitadas en principio: si se puede ejecutar en un contenedor debería funcionar correctamente en Kubernetes. Además, Kubernetes en ningún momento despliega código fuente o construye una aplicación. Esto se aplica directamente en los entornos de CI/CD, en donde los requisitos y las funciones se determinan por el equipo de trabajo.

En lo referente a servicios a nivel de aplicación, Kubernetes no ofrece mecanismos como *middlewares*, *frameworks* de procesamiento de datos, bases de datos, cache o sistemas de almacenamiento en clústers de forma nativa. Dichos componentes se pueden ejecutar en K8s pero se han de configurar mediante mecanismos externos.

Un factor muy importante es que no hay mecanismos de registro, monitorización o alertas: existen ciertas integraciones como prueba de concepto y mecanismos de recolección y exportación de métricas de funcionamiento.

¿Cómo funciona Kubernetes?

Cuando se despliega a Kubernetes, se crea un clúster. Un clúster se compone de un conjunto de máquinas (llamadas nodos) que ejecutan aplicaciones en contenedores. Es requisito que cada clúster contenga al menos un único nodo.

Dicho nodo aloja lo que se conoce como “pods”, un conjunto de componentes de la aplicación que se ejecutan en el nodo. Esto se apoya de un “control plane” que gestiona los nodos y los *pods en el clúster. Idealmente, en un entorno de producción, este *control plane* se ejecuta a lo largo de múltiples equipos y un clúster se despliega en múltiples nodos, ofreciendo una alta disponibilidad y tolerancia a fallos. Un entorno completo de Kubernetes se muestra en la figura 22:

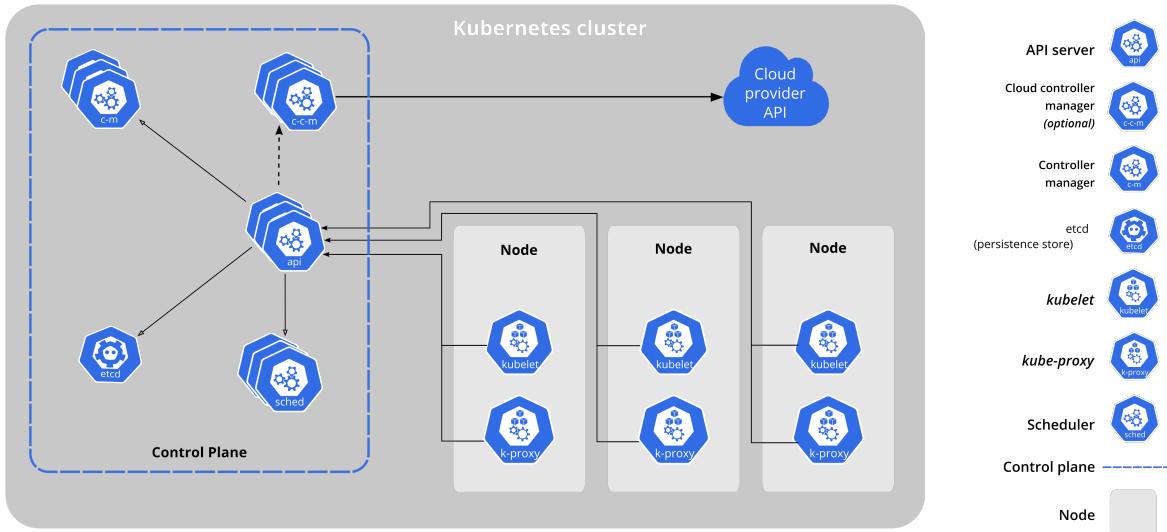


Figura 22: Componentes de Kubernetes [36].

El *control plane* de Kubernetes

Los componentes *control plane* permiten tomar decisiones globales en lo referente al clúster y detectar y responder a eventos que se den en el clúster.

De entre todos los componentes que existen, algunos importantes son:

- **kube-apiserver** es un componente que expone la API de Kubernetes. Está diseñado para escalar horizontalmente desplegando más instancias y se pueden ejecutar múltiples instancias para balancear la carga del sistema.
- **etcd** es un almacenamiento de alto rendimiento basado en clave–valor que se usa para almacenar los datos de los clústers.
- **kube-scheduler** monitoriza los *pods* que se han creado sin ningún nodo asignado a él y planifica su ejecución y en qué nodo se ejecutarán. Para definirlo se tienen en cuenta factores como los requisitos individuales y grupales de recursos, las características del *software*, *hardware* y de seguridad, la localización de los datos, etc.
- **kube-controller-manager** ejecuta los procesos *controller*. Estos controladores se definen según en donde se vayan a ejecutar:
 - Controladores de nodos que gestionan qué sucede cuando un nodo se cae.
 - Controladores de trabajo que gestionan cómo los *pods* ejecutan tareas.
 - Controladores de los *endpoints* que exponen dichos *endpoints* a los servicios que los necesiten.
 - Controladores de cuentas y tokens para la gestión del acceso a la API.
- **cloud-controller-manager** es similar a **kube-controller-manager** con la diferencia de que está destinado a la gestión de los controladores según el proveedor *cloud* que se esté usando.

Componentes de los nodos

Estos componentes se ejecutan en cada nodo y son el entorno de ejecución real de Kubernetes:

- kubelet es un agente que se ejecuta en cada nodo del clúster y se asegura de que los contenedores se estén ejecutando en un *pod*. Tomando los valores definidos en PodSpecs se asegura de que los contenedores descritos en esa especificación se estén ejecutando y no contengan fallos. Es importante tener en cuenta que kubelet solo gestiona contenedores creados dentro de Kubernetes.
- kube-proxy es un proxy de red que se ejecuta en cada nodo e implementa parte del concepto de servicio. Este componente gestiona las reglas de red de los nodos y aprovecha los mecanismos del sistema operativo para redirigir el tráfico. En otro caso, lo hace él mismo.
- *Container runtime* es la capa de ejecución de los contenedores en sí. Aquí aparecen tecnologías como Docker, containerd, etc.

Ejemplo aplicación en Kubernetes

Para completar esta sección, se va a exponer un ejemplo de cómo desplegar una aplicación en Kubernetes basándose en la guía publicada en la documentación oficial [37].

En este ejemplo se van a desplegar dos instancias de una aplicación “Hello World”, se va a crear un servicio que expone un puerto del nodo y se accederá a la aplicación en ejecución.

Nota: se asume que se tiene instalada la herramienta de gestión de Kubernetes kubectl y un servicio de Kubernetes local, como minikube.

Lo primero será definir la aplicación de Kubernetes que queremos desplegar. Al igual que con Docker Compose, la definición se basa en ficheros YAML (código 7):

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: hello-world
5 spec:
6   selector:
7     matchLabels:
8       run: load-balancer-example
9   replicas: 2
10  template:
11    metadata:
12      labels:
13        run: load-balancer-example
14    spec:
15      containers:
16        - name: hello-world
17          image: gcr.io/google-samples/node-hello:1.0
18          ports:
19            - containerPort: 8080
20              protocol: TCP

```

Listing 7: Configuración de una aplicación “Hello World” en Kubernetes.

En el fichero anterior se define, por una parte, la versión de la API de Kubernetes que se quiere usar. A continuación, el tipo de aplicación que se quiere desplegar (clave kind). Puede ser: Pod, DaemonSet, Deployment o Service. Después aparece la definición de las especificaciones del Pod que se va a desplegar. En este caso, se define un selector que va a ejecutar aplicaciones con una etiqueta “load-balancer-example”, se van a tener dos réplicas y se parte de una plantilla que define la etiqueta y el contenedor que va a ejecutar, en este caso una aplicación NodeJS que se expone en el puerto 8080.

Con el fichero ya definido se despliega la aplicación en el clúster:

```
1 kubectl apply -f hello-application.yaml
```

Esto habrá creado el despliegue (Deployment) y el conjunto de réplicas que contendrá los dos pods. Para obtener información sobre el despliegue:

```
1 kubectl get deployments hello-world
2 kubectl describe deployments hello-world
```

```
javinator9889@Tony-MkIII ~ ➔ kubectl get deployments hello-world
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
hello-world   2/2     2           2           76s
javinator9889@Tony-MkIII ~ ➔ kubectl describe deployments hello-world
Name:            hello-world
Namespace:       default
CreationTimestamp: Sun, 13 Jun 2021 13:28:19 +0200
Labels:          <none>
Annotations:    deployment.kubernetes.io/revision: 1
Selector:        run=load-balancer-example
Replicas:        2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=load-balancer-example
  Containers:
    hello-world:
      Image:      gcr.io/google-samples/node-hello:1.0
      Port:       8080/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type      Status  Reason
    ----      ----   -----
    Available  True    MinimumReplicasAvailable
    Progressing True    NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  hello-world-59966754c9 (2/2 replicas created)
Events:
  Type      Reason          Age   From           Message
  ----      ----   ----   ----   -----
  Normal   ScalingReplicaSet 87s   deployment-controller  Scaled up replica set hello-world-59966754c9 to 2
```

Figura 23: Información sobre los despliegues en Kubernetes.

A continuación, se crea un servicio que expondrá el despliegue anterior:

```
1 kubectl expose deployment hello-world --type=NodePort --name=example-
  ↵ service
2 kubectl describe services example-service
```

```
javinator9889@Tony-MkIII ~ ➤ kubectl expose deployment hello-world --type=NodePort --name=example-service
kubectl describe services example-service
service/example-service exposed
Name:           example-service
Namespace:      default
Labels:          <none>
Annotations:    <none>
Selector:       run=load-balancer-example
Type:           NodePort
IP Families:   <none>
IP:             10.108.12.79
IPs:            10.108.12.79
Port:           <unset>  8080/TCP
TargetPort:     8080/TCP
NodePort:       <unset>  32303/TCP
Endpoints:     172.17.0.3:8080,172.17.0.7:8080
Session Affinity: None
External Traffic Policy: Cluster
Events:         <none>
```

Figura 24: Información sobre el servicio recién desplegado de Kubernetes.

La salida anterior (imagen 24) nos indica información sobre lo que está en ejecución, el puerto de destino dentro del contenedor y el puerto expuesto en el valor NodePort. Para comprobar que la aplicación funciona, obtenemos la dirección IP del clúster de Kubernetes (comando `kubectl cluster-info`) y accedemos directamente desde el navegador o desde el terminal, a la dirección `http://<node-ip>:<node-port>` (figura 25):

```
X javinator9889@Tony-MkIII ➤ curl http://192.168.49.2:32303
Hello Kubernetes!
```

Figura 25: El clúster de Kubernetes responde a nuestra petición web.

Docker Swarm

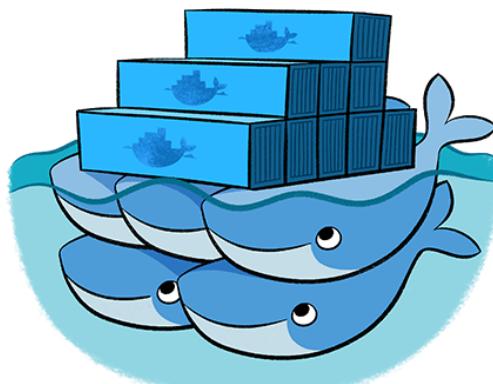


Figura 26: Logotipo de Docker Swarm [38].

Docker Swarm es la solución nativa de Docker para orquestación de contenedores. Cuando se hace una instalación de Docker automáticamente se ha instalado el gestor Swarm para manejar motores de Docker.

Esta solución es muy interesante ya que no hace falta instalar ninguna herramienta externa (como puede ser `kubectl`) sino que se trabaja directamente desde Docker CLI. Durante mucho tiempo esta herramienta no existía de forma nativa sino como un proyecto por separado, Docker “Classic” Swarm [39]. Sin embargo, tras la evolución de otras herramientas de gestión y la falta de desarrollo de la empresa encargada de dicho repositorio, en 2018 finalizó oficialmente su soporte y se pasó a trabajar de forma activa en SwarmKit [40]. Esta última herramienta se integra de forma nativa en el motor Docker desde la versión 1.12 en adelante.

Entre todas las características de Docker Swarm, algunas interesantes son [41]:

- Gestión de clústeres integrados directamente en el motor Docker, sin necesidad de *software* adicional para crear o gestionar un Docker Swarm.
- Diseño descentralizado: en lugar de definir quiénes son los nodos y sus roles en el momento del despliegue (como Kubernetes) se gestiona la especialización en tiempo de ejecución. Esto permite generar un clúster de Swarm desde una única imagen y luego decidir quién es quién.
- Modelo de servicios declarativo, en donde se define el estado deseado de varios servicios en el *stack* que conforma la aplicación. Por ejemplo, una aplicación puede estar definida por un servidor web que se comunica con otros servicios mediante una cola de mensajes y una base de datos como *backend*.
- Escalado: por cada servicio se especifican cuántas tareas se quieren desplegar. Cuando se hace un reescalado Docker Swarm se adapta automáticamente añadiendo o eliminando tareas para mantener el estado deseado.
- Gestión del estado deseado: el gestor Swarm monitoriza de forma activa los contenedores y mantiene el estado deseado cuando suceden inconvenientes. Por ejemplo, si se cuenta con un servicio con 10 réplicas y un *host* que tenía 2 falla, Docker Swarm creará dos nuevas réplicas para reemplazar a aquellas que ya no están, asignándolas a los nodos que estén disponibles.
- Red de múltiples *host*: mediante las redes *overlay*, que se vieron anteriormente, varios nodos se interconectan entre sí y se comunican como si no hubiera red de por medio. Docker Swarm se encarga de distribuir las direcciones y gestionar las redes.
- Descubrimiento de servicios: el gestor Swarm asigna un DNS único a cada servicio y hace balanceo de carga automático entre ellos, permitiendo el acceso individual a cada contenedor mediante el nombre DNS asignado.
- Balanceo de carga: se definen los puertos a exponer y automáticamente Docker Swarm distribuye la carga entre los nodos.
- Seguro por defecto, ya que las comunicaciones utilizan autenticación mútua y cifrado basados en TLS.
- Actualizaciones continuas: se puede realizar una actualización incremental de cada una de las réplicas y, si en algún momento algo va mal, se puede recuperar la versión anterior.

¿Cómo funciona Docker Swarm?

La idea principal detrás de este mecanismo de gestión de contenedores es su integración directa dentro de Docker. Esto se consigue mediante el uso de la herramienta SwarmKit, la cual es un proyecto por separado de Docker que se integra en el mismo e implementa la capa de orquestación de Docker.

Por ende, un *swarm* consiste en múltiples *hosts* Docker que se ejecutan en modo Swarm y actúan como gestores, esto es, definen la delegación y los miembros; o como trabajadores, que ejecutan los servicios de Swarm. Ambos roles no son excluyentes, por los que un *host* Docker puede actuar de las dos maneras a la vez.

Al definir un Swarm se especifica el estado ideal del mismo: número de réplicas, recursos de memoria y red disponibles, puertos expuestos, etc. Mientras se esté ejecutando, Docker intenta mantener ese estado deseado. Por ello, cuando un nodo ya no está activo se planifican las tareas pendientes en otro nodo (como se vio anteriormente).

Uno de los factores diferenciales principales de Docker Swarm es que se pueden actualizar las configuraciones de los servicios (incluidas redes y almacenamiento) sin la necesidad de recrear el servicio: automáticamente, aquellos contenedores con una configuración anterior serán reemplazados por los nuevos que cumplan con la nueva configuración. Por otra parte, que un anfitrión Docker esté en modo Swarm no impide que se puedan ejecutar contenedores de forma estándar en el mismo, lo cual brinda una gran potencia a este mecanismo.

Es importante recalcar que la definición de los servicios Swarm se hace a partir de un fichero de Docker Compose: si se tiene un conjunto de contenedores definidos en un fichero `docker-compose.yml` versión 3.0 o superior, se puede desplegar una pila Swarm desde ese mismo fichero sin necesidad de ninguna configuración extra.

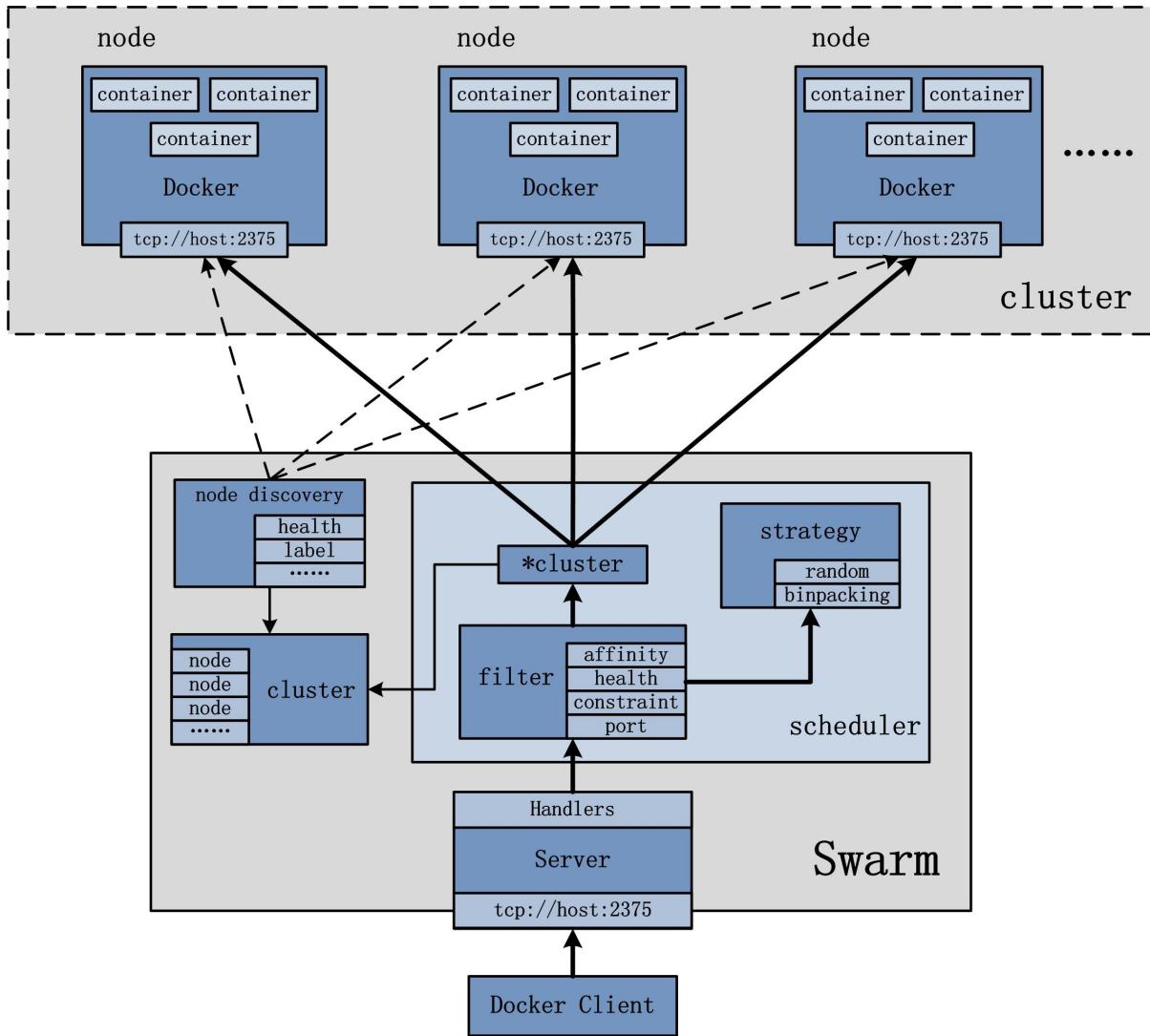


Figura 27: Arquitectura de Docker Swarm [42].

En la figura 27 se define cómo sería la arquitectura de Docker Swarm en un entorno ya desplegado. Es importante fijarse en que el sistema, en comparación con Kubernetes, es muy similar: hay un punto de entrada, que en Kubernetes era el *control plane*, y a continuación el motor de gestión de Swarm, que distribuye las tareas entre contenedores y se encarga de gestionar y monitorizar tanto a los nodos como a los recursos. Después, cada motor Docker tendrá su aplicación en ejecución y ofrecerá el servicio cuando lo solicite [43].

Los nodos en Swarm

Un nodo es una instancia del motor Docker que participa en el clúster. En un mismo equipo se pueden ejecutar múltiples nodos Docker, pero en un entorno de producción estarán distribuidos a lo largo de un conjunto de múltiples máquinas físicas y en la nube.

Cuando se despliega una aplicación en el Swarm se envía el servicio al nodo gestor. Dicho nodo genera unidades de trabajo (tareas) y las reparte entre los nodos trabajadores. Por otra parte, dichos nodos también definen la orquestación y las funciones de gestión del clúster requeridas para mantener el estado deseado.

De esta forma, los nodos trabajadores reciben y ejecutan las tareas recibidas por el nodo gestor. Por defecto, todos los nodos son nodos trabajadores pero se puede configurar y especificar que un nodo en particular sea únicamente un gestor. Cuando un nodo trabajador finaliza su tarea, notifica al nodo gestor para que este pueda perseverar el estado del sistema como se espera [43].

Los servicios y las tareas

Un servicio es una definición de tarea que se ejecuta en un nodo trabajador dentro de un clúster Swarm, definiendo así la estructura principal de Docker Swarm y la interacción principal de un usuario con el mismo.

Los servicios al final son imágenes Docker con un comando asociado a los mismos, que luego se convertirán en contenedores en ejecución. En particular, un servicio puede estar replicado y será distribuido entre varias réplicas; o bien el servicio es global y se ejecutará en todos los nodos Swarm [43].

Balanceo de carga

Una de las características principales de Docker Swarm es el uso de “Ingress” como gestor de la carga para exponer los servicios y repartir el trabajo entre los nodos. Esto se consigue asignando manualmente un puerto donde se quiera publicar la aplicación o bien se designará automáticamente.

De esta forma, proveedores externos acceden directamente al puerto expuesto y Swarm gestionará las peticiones entre los distintos servicios, de manera que en apariencia se accede siempre al mismo nodo. Internamente se usa un servicio DNS que se asigna automáticamente a cada tarea y, gracias al balanceador de carga nativo que tiene, distribuye las peticiones.

Ejemplo de aplicación en Docker Swarm

Una de las grandes ventajas que presenta Docker Swarm es que utiliza la sintaxis de Docker Compose para definir sus servicios, con algunos añadidos no soportados por el mismo.

Para este ejemplo, se va a desplegar una instancia de GitLab CE con hasta cuatro réplicas para operaciones de CI/CD automatizadas. El fichero que lo define es (código ??):

```
1 version: "3.6"
2 services:
3   gitlab:
4     image: gitlab/gitlab-ce:latest
5     ports:
6       - "22:22"
7       - "80:80"
8       - "443:443"
9     volumes:
10      - $GITLAB_HOME/data:/var/opt/gitlab
11      - $GITLAB_HOME/logs:/var/log/gitlab
12      - $GITLAB_HOME/config:/etc/gitlab
13     environment:
```

```

14     GITLAB_OMNIBUS_CONFIG: "from_file('/omnibus_config.rb')"
15   configs:
16     - source: gitlab
17       target: /omnibus_config.rb
18   secrets:
19     - gitlab_root_password
20 gitlab-runner:
21   image: gitlab/gitlab-runner:alpine
22   deploy:
23     mode: replicated
24     replicas: 4
25 configs:
26   gitlab:
27     file: ./gitlab.rb
28 secrets:
29   gitlab_root_password:
30     file: ./root_password.txt

```

Listing 8: docker-compose.yml para una instancia de GitLab y 4 runners.

En el fichero anterior se puede ver que la estructura es muy similar a la de Docker Compose. Sin embargo, aparecen algunas claves nuevas que no existían con anterioridad:

- services define los servicios que conformarán la aplicación.
- deploy define el modo de despliegue (mode) y la cantidad de réplicas que se usarán (replicas). Si no se especifica, se usa una única réplica.
- configs define de forma común la configuración que tendrán las réplicas.
- secrets define contraseñas o información sensible que necesitarán las réplicas en tiempo de ejecución.

Aunque en este caso se haya usado un fichero docker-compose.yml adaptado para Docker Swarm, se podría perfectamente haber usado el fichero descrito en el apartado 2.4 en el que se desplegaba una aplicación Wordpress con NGINX y MySQL.

Se definen los ficheros necesarios y se pasa a desplegar el servicio (figura 28):

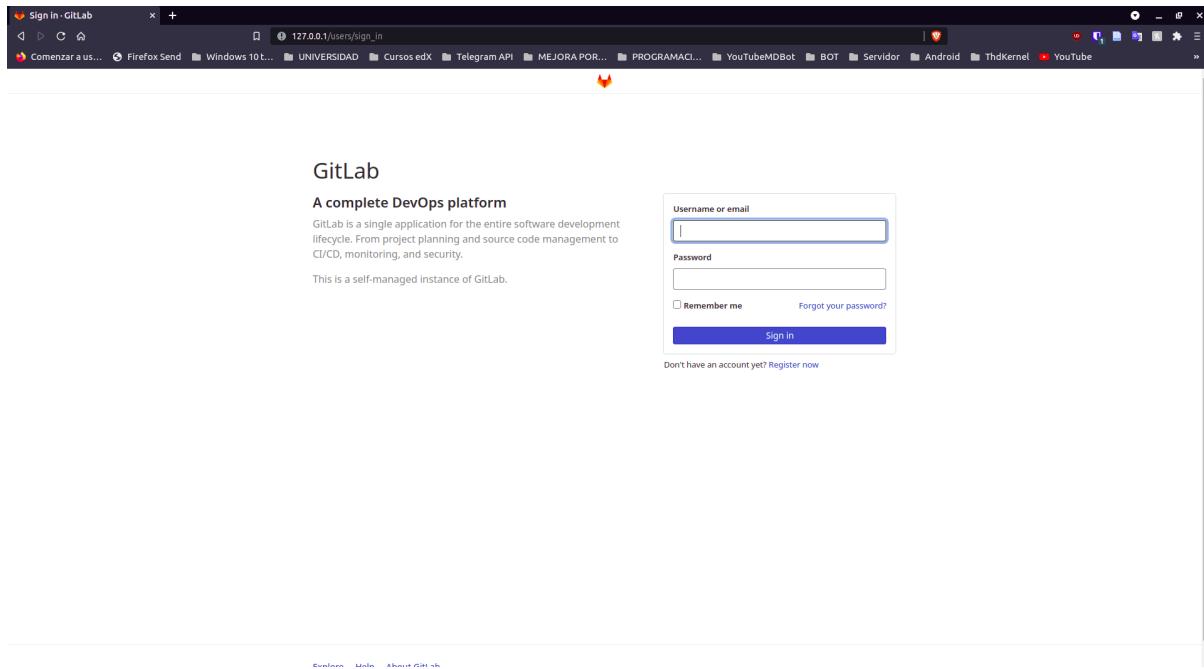
```
1 docker stack deploy --compose-file docker-compose.yml gitlab-stack
```

```

javinator9889@Tony-MkIII ~ /gitlab > ls -l
total 12
-rw-rw-r-- 1 javinator9889 javinator9889 674 jun 13 17:48 docker-compose.yml
-rw-rw-r-- 1 javinator9889 javinator9889 87 jun 13 17:48 gitlab.rb
-rw-rw-r-- 1 javinator9889 javinator9889 21 jun 13 17:48 root_password.txt
javinator9889@Tony-MkIII ~ /gitlab > cat gitlab.rb
| File: gitlab.rb
1 | gitlab_rails['initial_root_password'] = File.read('/run/secrets/gitlab_root_password')
javinator9889@Tony-MkIII ~ /gitlab > cat root_password.txt
| File: root_password.txt
1 | SecretRootPassword:)
javinator9889@Tony-MkIII ~ /gitlab > docker stack deploy --compose-file docker-compose.yml gitlab-stack
Creating network gitlab-stack_default
Creating secret gitlab-stack_gitlab_root_password
Creating config gitlab-stack_gitlab
Creating service gitlab-stack_gitlab
Creating service gitlab-stack_gitlab-runner
javinator9889@Tony-MkIII ~ /gitlab >

```

Figura 28: Despliegue del Swarm con GitLab en su interior.



2.6. Líneas futuras de desarrollo e innovación

Docker es una tecnología muy en boga hoy en día. Sin embargo, varios problemas durante el desarrollo y la falta de actualizaciones con características demandadas han llevado al “rey” a caerse.

Parece que en los últimos meses (a junio de 2021) se han puesto las pilas y han retomado con fuerza el desarrollo de su motor de ejecución y herramienta de orquestación. Uno de los problemas que se encuentran muchas empresas a la hora de desplegar contenedores es su naturaleza efímera y no persistente que, si bien es cierto es su gran ventaja, muchas veces puede jugar en su contra.

Sería necesario un modelo de contenedor sostenible en el tiempo, que se mantenga por sí mismo y que no requiera de atención por parte del gestor. Parece ser que cada vez la tendencia tiene más hacia allí y Docker está enfocando sus esfuerzos en escuchar a la comunidad y mantenerse como líder.

Otro factor crítico fue la noticia de *deprecation* de Docker dentro de Kubernetes [44]: en favor del estándar de contenedores containerd las imágenes nativas de Kubernetes se sustituyen por aquellas basadas en CRI (*Container Runtime Interface*). Si bien esto no deja de lado a Docker ya que se sigue pudiendo ejecutar en Kubernetes, es un golpe duro ya que las empresas están centrando sus esfuerzos en otras tecnologías de contenedores.

Es por ello por lo que Docker está apostando en su tecnología de orquestación, Swarm, para intentar imponerse como nuevo líder en gestión de contenedores. Es una tarea compleja (principalmente porque Kubernetes es el líder) pero tiene muchos factores a su favor, como la fácil integración y la sencillez en las operaciones.

3. Seguridad en Docker

A lo largo de todo el documento se ha visto cómo la tecnología de contenedores ha enfocado gran parte de su esfuerzo en la seguridad: desde el diseño inicial hasta la implementación final las características de seguridad de los contenedores han supuesto un punto de inflexión en estas tecnologías.

La cuestión es: ¿cuánto de seguros son? En común todas las tecnologías tienen un “fallo” de base: requieren de permisos de administrador para ejecutarse. Eso o bien el usuario que gestiona los contenedores pertenece a un grupo privilegiado, que sería equivalente a ejecutar “todos” los comandos como administrador.

En esta sección se va a tratar por una parte la seguridad en Docker desde el punto de vista de más bajo nivel, como es la comunicación con el kernel; hasta aspectos más “elevados” como son diferencias con otras soluciones como chroot, cómo afecta el *firewall* a la seguridad de los contenedores y cómo de seguras son las comunicaciones entre contenedores.

3.1. Análisis de la pila Docker

Anteriormente, en la imagen 18 se muestra cómo el motor Docker interactúa con el kernel de Linux para proveer aislamiento. Entre otras librerías, Docker usa en particular los *cgroups* [45] y los *namespaces* de Linux [46].

Sendas funcionalidades del kernel ofrecen una gran capa de seguridad y aislamiento de procesos de forma nativa en todas las máquinas Linux, y Docker aprovecha esa infraestructura para proteger los contenedores a un nivel muy bajo: a nivel de kernel.

Este planteamiento ya augura un buen presagio en tanto que no se usan aplicaciones externas o librerías de terceros para el aislamiento y protección de capas sino que se usa una arquitectura a bajo nivel ampliamente depurada y probada como es el kernel de Linux. Además, esto permite también una gran portabilidad, ya que el “único requisito” para ejecutar Docker sería contar con un kernel de Linux.

A la hora de hablar o revisar la seguridad de Docker, existen cuatro áreas primordiales en las que indagar [47]:

- La seguridad intrínseca del kernel así como su soporte para *namespaces* y *cgroups*.
- La superficie de ataque sobre el servicio de Docker en sí.
- Lagunas en las configuraciones de un contenedor o bien por defecto o bien introducidas por el usuario.
- Las características de seguridad del kernel y su interacción con Docker.

Linux kernel *namespaces*

A nivel de funcionamiento, Docker es similar a los contenedores LXC y comparten los mismos mecanismos de seguridad. Por ende, cuando se crea un contenedor con el comando

`docker run` internamente se están creando un conjunto de *namespaces* y *cgroups* para el contenedor.

El primero ofrece la primera y mayor forma de aislamiento – un proceso que se ejecuta dentro de un *namespace* es incapaz de ver (y mucho menos afectar) a otros procesos en ejecución en otro contenedor o en la máquina anfitriona.

Esto llega al nivel de que cada contenedor tiene su propia pila de protocolos de red, lo cual implica que un contenedor nunca tendrá acceso privilegiado a *sockets* o interfaces de otro contenedor o del sistema anfitrión. Sin embargo, si se configura correctamente un sistema anfitrión las comunicaciones entre contenedores pueden existir perfectamente, a través de la misma interfaz de red. Si por el contrario se expone el puerto del contenedor entonces otros equipos podrán enviarle mensajes o *pings* al contenedor, paquetes TCP/UDP y establecer distintos tipos de conexiones. Sin embargo, se pueden restringir si es necesario: a fin de cuentas, la interfaz por defecto que usan los contenedores es la tipo puente, que implica que en apariencia son equipos conectados a un *switch*.

Por otra parte, los *namespaces* de Linux tienen la ventaja de que son prestaciones probadas y testeadas a lo largo del tiempo, introducidas en el año 2008 en la versión 2.6.15. Además, su diseño y arquitectura se basa en un intento de mejora sustancial de OpenVZ [48], desarrollado en 2005.

En un estudio realizado en el año 2018 [49], se detectaron ciertos problemas residentes en el kernel que impedían a los contenedores hacer uso de características avanzadas de seguridad del kernel: no se pueden aplicar políticas para comprobar la integridad, regular la ejecución de código, control de acceso, etc. que permiten prevenir problemas de seguridad referentes a la aplicación. Ante el intento de añadir nuevas opciones al kernel que permitan el acceso a dichos recursos, se descartaron debido a ser solución *ad-hoc* que suponían muchas veces una brecha de seguridad más allá de una opción real.

Sin embargo, en dicho estudio se proponen los *security namespaces*, una característica del kernel que permitiría la ejecución de contenedores con la totalidad de las prestaciones disponibles produciendo una latencia menor al 0,7 % en las llamadas al sistema.

Esta característica ha seguido evolucionando hasta nuestros días en la forma de los *user namespaces*, que son los usados e implementados actualmente por Docker. Un problema común del usuario *root* en Linux no es que sea necesariamente el administrador sino las posibilidades que tiene (en particular, las Linux kernel *capabilities* [50]): estas capacidades se pueden otorgar o bien mediante un escalador de privilegios (por ejemplo, usar el comando *sudo*) o mediante el ajuste de permisos, como el SUID o un cambio de *namespace*. Esto se aprovecha por el motor Docker para definir qué posibilidades tiene un contenedor, que pueden ser añadidas o restringidas por el usuario.

El problema es cuando un usuario define un contenedor que requiere de más capacidades de las que él mismo tiene. Por defecto, los contenedores se ejecutan siempre como *root*, por lo que la situación anterior es perfectamente posible, y conlleva posibles abusos en forma de fallo de seguridad.

Con el uso de los *user namespaces* este problema se puede abordar fácilmente: un *namespace* es a fin de cuentas un “mapa” en donde un UID y GID virtuales se mapean con los UID/GID reales y se exponen en la ruta /proc [51]. El mapeo presenta la siguiente forma (figura 29):

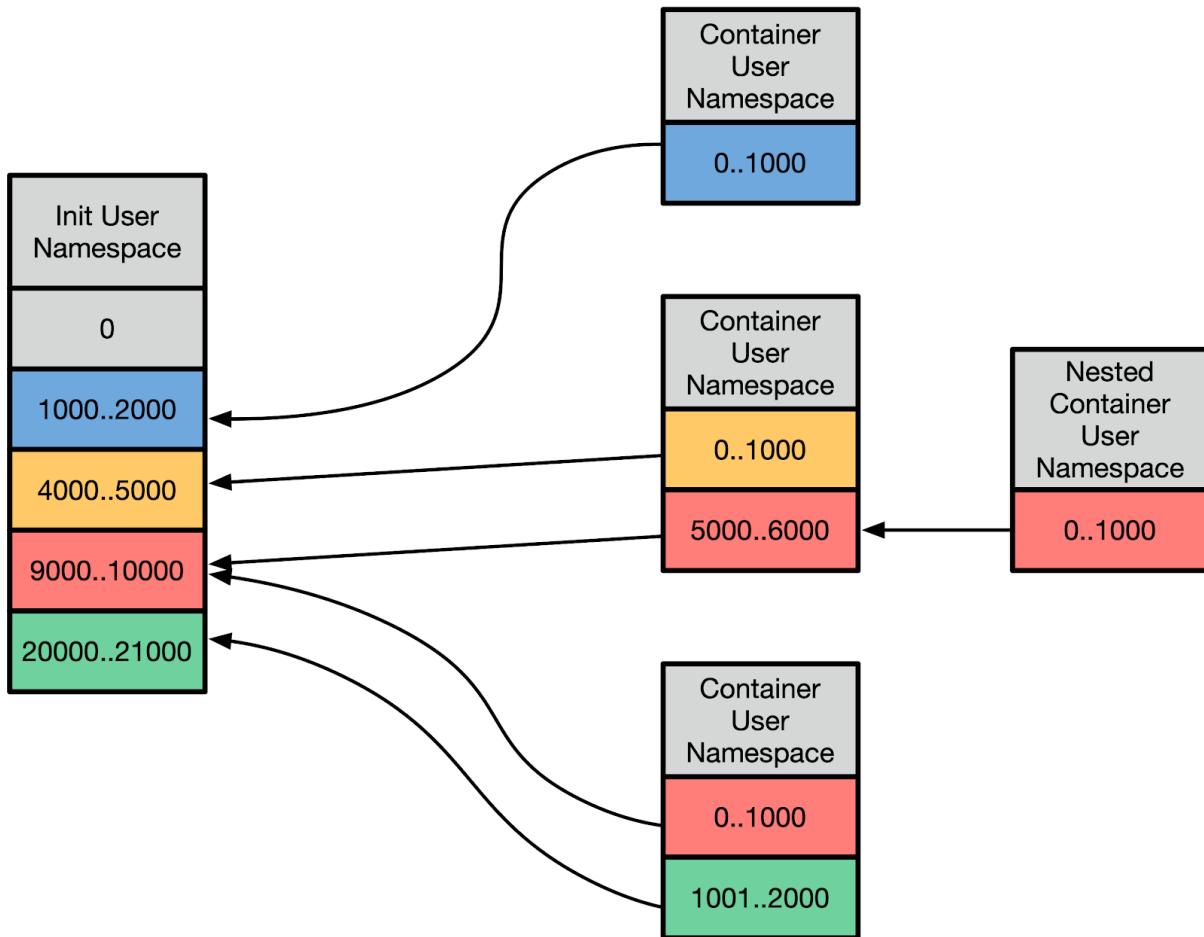


Figura 29: Mapeo de los UID del *namespace* del contenedor al *namespace* del sistema [51].

¿Por qué es potente esta característica del sistema?:

- Permite, por una parte, definir ciertos UIDs únicamente dentro del contenedor. De esta forma, si un UID no está relacionado con un UID real del equipo anfitrión, al intentar examinar un fichero con dicho UID aparecerá un error del tipo `overflowuid` en los ficheros de `/proc` [52].
- Desde el punto de vista de un *namespace* de usuario el contenedor se ejecuta con UID 0 cuando en realidad está usando un rango de valores mapeados en dicho *namespace*.
- Los subsistemas Linux pueden ejecutar la función “`ns_capable`” (que permite definir si una tarea tiene en realidad mayores capacidades) usando un *namespace* específico de un recurso. De esta forma, los procesos pueden realizar acciones “privilegiadas” sin tener en realidad privilegios sobre el sistema anfitrión.

En la actualidad, el servicio Docker inicia los contenedores siempre con las capacidades mínimas posibles [47]: por ejemplo, servidores web que únicamente necesitan exponer un puerto protegido (< 1024) se le garantiza el permiso `net_bind_service` en lugar de ejecutarse como administrador.

Esto permite que diversos procesos que siempre se ejecutan como administrador (SSH, cron, módulos del kernel, ...) usen un usuario con menos privilegios y se les garantice únicamente

acceso a las características que necesitan. Algunas operaciones además son directamente gestionadas por Docker en lugar de por las aplicaciones que se ejecutan:

- El acceso mediante SSH se realiza mediante un único servidor gestionado por Docker.
- Los procesos cron se ejecutan como usuario estándar, cuando es posible.
- La gestión de logs se delega a Docker u otro servicio.
- La gestión del hardware es irrelevante, lo que se traduce en que instrucciones como udevd nunca serán necesarias.
- La gestión de las redes sucede fuera de los contenedores, lo cual implica que un contenedor nunca tendrá que ejecutar tareas con ifconfig, route o ip.

La implicación directa es que los contenedores no necesitan permisos de administrador reales (al menos, no todos) y se restringen las posibilidades del mismo. Por ende, se pueden aplicar algunas medidas de seguridad que impliquen denegar todas las operaciones de montaje, impedir el acceso a sockets, prohibir ciertas operaciones sobre discos (particionado, cambiar permisos, etc.), denegar la carga de módulos del kernel y demás.

Así, si un intruso es capaz de acceder a un contenedor y escalar privilegios hasta ser root, el daño que podrá hacer sobre el sistema estará limitado. Pero esto siempre ha de ir acompañado de una gestión competente: cuando se crea un contenedor, lo ideal sería quitar todas aquellas capacidades del contenedor que no sean necesarias o no se vayan a usar.

Linux control groups

Los grupos de control de Linux no son necesariamente una característica de seguridad sino de control y regulación de acceso a los recursos. En el estudio mencionado anteriormente ([49]) uno de los problemas que se vieron en comparación con las máquinas virtuales era la restricción del acceso a los recursos y la limitación de ciertas acciones.

Durante bastante tiempo los contenedores Docker no contaban con una limitación en la cantidad de recursos disponibles hasta que se adaptó el motor de ejecución para trabajar con los cgroups, que existen en Linux desde el 2006 (versión 2.6.24 [47]).

Esta característica se incluye en el apartado de seguridad porque permite proteger a un contenedor de ataques de denegación de servicio, “molestar” a otro contenedor, etc.

Ataque al servicio de Docker

La ejecución de contenedores implica el uso del servicio de Docker para ello. Si no se ha optado por el modo “sin privilegios”, dicho servicio se ejecutará siempre como root, por lo que será uno de los elementos más atacados y vulnerables que existan.

En general, sobre el servicio Docker no se pueden tomar muchas acciones en lo referente a privilegios o código fuente pero sí se pueden aplicar ciertas restricciones sobre el uso. Lo primero es restringir qué usuarios pueden interactuar con el servicio de Docker, ya que es una implicación directa de seguridad. En Docker, se puede hacer *bind mounts* del directorio raíz

del sistema (/) dentro de un directorio /host en el contenedor. Desde allí, se puede acceder a cualquier fichero, directorio o dispositivo sin restricciones ni limitaciones.

Es por este motivo por el que Docker CLI utiliza una API REST contra un socket UNIX en lugar de usar la dirección 127.0.0.1, ya que permite ajustar los permisos del socket usando el sistema de permisos de Linux. Bajo esta premisa, si se decidiese crear un contenedor que expusiera una API REST bajo HTTP habría que dedicar especial atención a qué ficheros se puede acceder y controlar todavía más los parámetros que se reciben. Inclusive aunque se restrinja el acceso a la API únicamente a la red local mediante el uso de *firewalls*, otros contenedores podrían comprometer el sistema. Por ello, es obligatorio proteger los *endpoints* usando HTTPS y certificados digitales, y altamente recomendable que solo sea accesible a través de una VPN.

Por otra parte, el servicio Docker es potencialmente vulnerable a otras entradas como la carga de una imagen desde el disco con `docker load` o desde la red con `docker pull`. Por ello, el primer paso desde la versión 1.10.0 de Docker es la de crear una jaula chroot en donde desempaquetar la imagen y preparar el entorno. Para evitar estos problemas se puede configurar el servidor Docker para que únicamente trabaje con imágenes firmadas – DCT (*Docker Content Trust*).

Esta característica es nativa al servicio de Docker y se puede usar directamente desde Docker CLI. Mediante esta herramienta, un usuario que publique una imagen podrá firmarla y darle la seguridad a los clientes de que se está usando una imagen oficial. Las claves de firma se disgregan en tres tipos de claves:

- Claves offline, para la firma inicial de la imagen asociada a un usuario.
- Clave de etiquetado (*tag key*), asociada a un repositorio de imágenes.
- Clave de tiempo (*timestamp key*), asociada a un repositorio y creada por Docker.

Todas las claves se generan a raíz de la clave offline, también conocida como “*root key*” (figura 30):

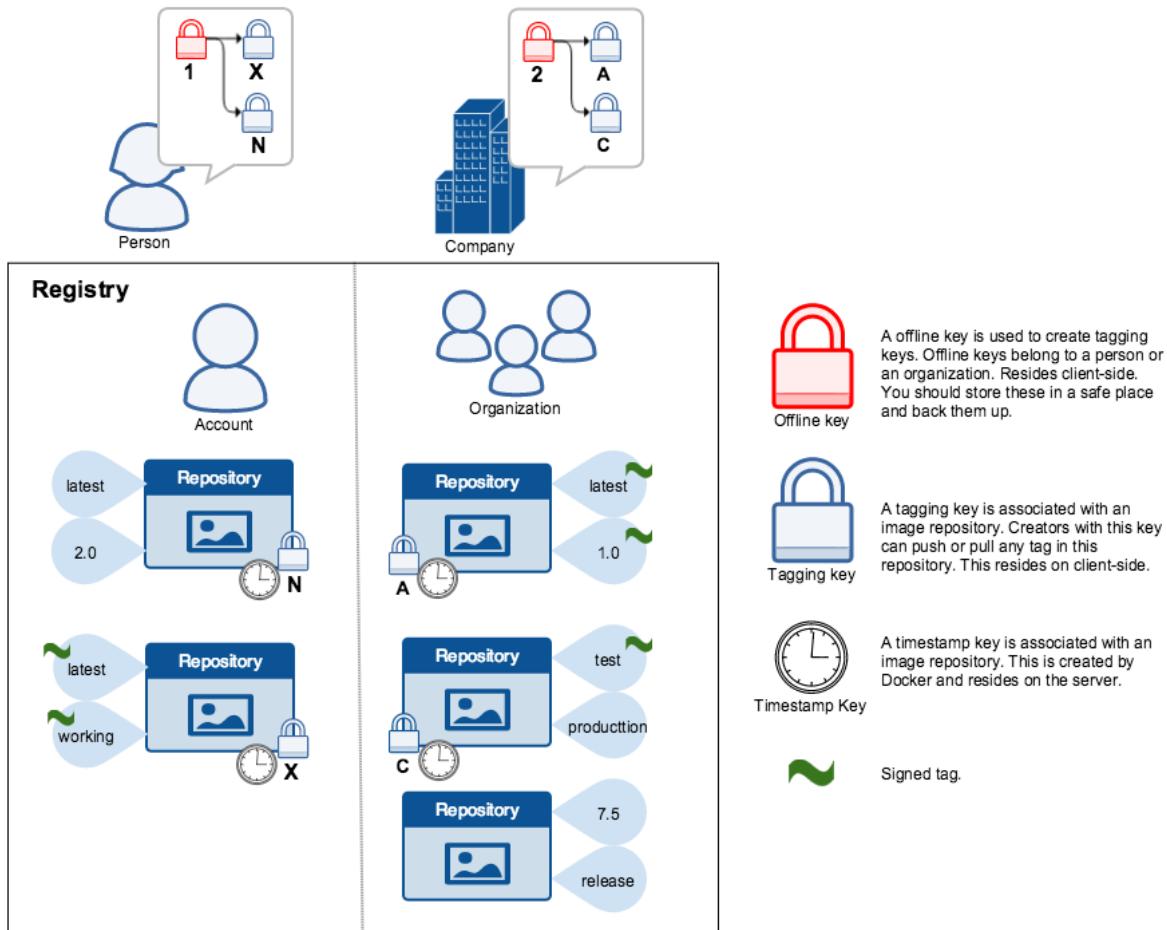


Figura 30: Distribución de las claves para la firma de imágenes en Docker [53].

Mayor protección de Docker

Una de las características fundamentales de que se ejecute sobre el kernel de Linux es que Docker está directamente integrado con otras herramientas de seguridad.

Aunque el motor Docker deshabilite ciertas posibilidades sobre los contenedores esto no implica que otros sistemas no se puedan usar en conjunto. Por ejemplo:

- Se puede ejecutar un kernel con GRSEC [54] y PAX [55] para añadir opciones de seguridad tanto en la compilación como en tiempo de ejecución. Por otra parte, mediante el uso de técnicas de seguridad como la aleatorización de direcciones se mitigan muchas vulnerabilidades del sistema. Lo interesante es que no es necesaria ninguna interacción por parte de Docker ya que son características que se aplican a lo largo de todo el sistema.
- Si la aplicación viene con un modelo de seguridad que use AppArmor o SELinux se puede integrar directamente dentro de Docker, sin necesidad de hacer ningún cambio. Esto otorgará una capa extra de seguridad que se complementa con la restricción de las capacidades del contenedor.

En principio, cualquier herramienta externa se podría usar dentro de un contenedor Docker para proteger el acceso y añadir más capas de seguridad. Esto se hace principalmente sobre

sistemas de ficheros en red o compartidos con otros contenedores, donde la seguridad es crítica.

3.2. Diferencias fundamentales con chroot

Desde las primeras versiones de Linux, el concepto de “enjaular” ya existía. Crear un chroot se traduce en que una aplicación se ejecuta sobre un directorio definido como si dicho directorio fuese la raíz. Por ejemplo, se crea un chroot en la ruta /home/user/jail. Cualquier proceso que se ejecute dentro de dicha ruta pensará que se está ejecutando sobre la raíz /, y no tendrá acceso al sistema de ficheros que haya por encima.

Esta característica fue uno de los primeros pasos en la creación y configuración de *sandboxes*, y además añadía cierta portabilidad porque siempre se podía comprimir dicho directorio, llevarlo a otro equipo y trabajar sobre sus contenidos directamente. Además, como en ese directorio el árbol UNIX no se respeta del todo, es posible aislar un proceso del resto excluyendo el directorio /proc o /run de la jaula.

¿Qué diferencias existen con Docker? A lo largo de todo el documento, ya se tiene una idea muy clara de qué permite Docker. Principalmente, Docker presenta un mecanismo de aislamiento mucho más sofisticado. En una jaula es necesario renunciar a ciertos permisos o características para poder trabajar con seguridad. A veces es posible que no sea sencillo ejecutar un proceso o aplicación en una jaula y la configuración que se diseñe podría dañar directamente al sistema, ya que una jaula siempre se ejecuta como administrador. Por su parte, el aislamiento con Docker se consigue mediante los *namespaces* de Linux.

Por otra parte, en Docker los procesos tienen su propia tarjeta de red, su propio identificador (*hostname*), memoria dedicada, limitación de recursos, ... Es decir, es una solución mucho más sofisticada.

Docker se compara más a una máquina virtual ya que permite usar un sistema distinto al que está instalado, mientras que en una jaula se comparte todo del sistema anfitrión. Por ende, la característica principal que diferencia una solución de otra es la capacidad de aislar procesos, recursos, memoria y red.

3.3. Seguridad en las comunicaciones de red – *firewall*

Internamente, Docker utiliza un *firewall* para aislar las comunicaciones de red. En particular, el *firewall* empleado es *iptables* [56]. Esta herramienta está directamente embebida dentro del kernel de Linux, lo cual asegura una gran compatibilidad universal y un gran rendimiento, ya que trabaja directamente sobre el *hardware* cuando es posible.

Cuando se configura un contenedor o una red, el servicio de Docker inserta las reglas necesarias en las tablas de *iptables*. Esto se realiza sobre el sistema, por lo que es posible que exista conflicto con reglas que puedan existir ya.

Sin embargo, esto abre nuevas opciones: configurar tus propias reglas para restringir el acceso a ciertos contenedores. Docker tiene su propio conjunto de reglas (DOCKER) y luego tiene un grupo de reglas destinadas al usuario (DOCKER-USER). En este último un administrador puede definir qué reglas quiere aplicar a los contenedores.

La versatilidad de esto permite, por una parte, definir que las conexiones al servicio de Docker solo se realicen desde una red interna:

```
1 sudo iptables -I DOCKER-USER -m iprange -i <external-interface> ! --src-
   ↳ range 192.168.1.1-192.168.1.3 -j DROP
```

Listing 9: Restricción de las conexiones a Docker según un rango de IPs [56].

Si, por otra parte, Docker actúa como si fuese un router, se pueden permitir ciertas conexiones desde una interfaz hasta otra:

```
1 sudo iptables -I DOCKER-USER -i <source-interface> -o <destination-
   ↳ interface> -j ACCEPT
```

Una característica interesante es que se puede combinar con otros sistemas de *firewall* como `firewalld` y, en un futuro, se plantea adoptar `nftables` como nuevo gestor de peticiones de red (es el sustituto de `iptables` en las nuevas versiones de Linux).

3.4. Conclusiones

Los contenedores, en apariencia, son muy seguros. Si se toman las medidas pertinentes y se ejecutan los procesos como un usuario sin privilegios, es muy complejo que pueda haber un fallo de seguridad usando esta tecnología.

Si además se combina lo anterior con una capa extra de seguridad como puede ser AppArmor, SELinux, GRSEC y reglas de *firewall* que gestionen las comunicaciones, se tiene un compartimento estanco en donde desplegar cómodamente las aplicaciones, de una forma fácil, sencilla y funcional.

En diversos estudios se ha visto cómo Docker ofrece una capa de seguridad muy densa tanto por encima de las aplicaciones como por debajo, a nivel de kernel. Sin embargo, se han detectado varios fallos de seguridad en lo referente a las interfaces que pueden ser aprovechados para hacer ARP *spoofing* y MAC *flooding*. Esto se debe a que principalmente no añade ningún tipo de filtro al tráfico de red que circula por la interfaz [57]. Sin embargo, esto se puede subsanar añadiendo reglas de *firewall* que regulen y filtren los paquetes que circulan por la interfaz.

Otro problema podría surgir de ejecutar los contenedores en modo privilegiado, lo cual implica que es casi como si se estuviera ejecutando en la máquina anfitriona con privilegios root. Aunque hay que tener en cuenta que esta característica casi nunca se usa debido a sus implicaciones de seguridad y a que la gran mayoría de contenedores no requieren de esta opción para funcionar.

Se ve que efectivamente el aislamiento que ofrece Docker es muy eficaz y, combinado con unas buenas prácticas, explica la gran evolución y crecimiento de este tipo de tecnologías durante los últimos años. Eso combinado con la gran velocidad en el desarrollo y en la ejecución que presenta lo ha convertido en la “opción ganadora” que se ha mantenido al alza durante los últimos 4 años.

Referencias

- [1] «History of Technology Timeline,» Encyclopedia Britannica. (), dirección: <https://www.britannica.com/story/history-of-technology-timeline> (visitado 07-05-2021).
- [2] «Evolution of Data Storage Timeline,» The Gateway. (), dirección: </gateway/data-storage-timeline/> (visitado 07-05-2021).
- [3] WeComputingTech. «Storage devices london | We Computing Blog.» () , dirección: <http://www.wecomputing.com/blog/tag/storage-devices-london/> (visitado 07-05-2021).
- [4] «How To Become A Web Developer in 2021 – Everything You Need To Know.» (), dirección: <https://careerfoundry.com/en/blog/web-development/what-does-it-take-to-become-a-web-developer-everything-you-need-to-know-before-getting-started/> (visitado 07-05-2021).
- [5] *Dependency hell*, en Wikipedia, 29 de mayo de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Dependency_hell&oldid=1025704309 (visitado 03-06-2021).
- [6] «Docker overview,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/get-started/overview/> (visitado 03-06-2021).
- [7] «What is a Container? | App Containerization | Docker.» (), dirección: <https://www.docker.com/resources/what-container> (visitado 03-06-2021).
- [8] «Containerd.» (), dirección: <https://containerd.io/> (visitado 03-06-2021).
- [9] «Container Runtime with Docker Engine | Docker.» (), dirección: <https://www.docker.com/products/container-runtime> (visitado 03-06-2021).
- [10] S. Yegulalp. «What is Docker? The spark for the container revolution,» InfoWorld. (), dirección: <https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html> (visitado 03-06-2021).
- [11] «Docker Desktop WSL 2 backend,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/docker-for-windows/wsl/> (visitado 03-06-2021).
- [12] S. Kulshrestha. «Docker Networking – Explore How Containers Communicate With Each Other,» Medium. (10 de sep. de 2020), dirección: <https://medium.com/edureka/docker-networking-1a7d65e89013> (visitado 03-06-2021).
- [13] S. Watts. «The State of Containers Today: A Report Summary,» BMC Blogs. () , dirección: <https://www.bmc.com/blogs/state-of-containers/> (visitado 04-06-2021).
- [14] «6 Container Adoption Trends of 2020,» StackRox: Kubernetes and container security solution. (), dirección: <https://www.stackrox.com/post/2020/03/6-container-adoption-trends-of-2020/> (visitado 04-06-2021).
- [15] «Container Adoption Statistics: The Future of the Container Market,» Capital One. (), dirección: <https://www.capitalone.com/tech/cloud/container-adoption-statistics/> (visitado 04-06-2021).
- [16] «Download the 2018 Docker Usage Report,» Sysdig. (29 de mayo de 2018), dirección: <https://sysdig.com/blog/2018-docker-usage-report/> (visitado 04-06-2021).
- [17] «Manage data in Docker,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/storage/> (visitado 04-06-2021).

- [18] *UnionFS*, en *Wikipedia, la enciclopedia libre*, 2 de jul. de 2020. dirección: <https://es.wikipedia.org/w/index.php?title=UnionFS&oldid=127421790> (visitado 04-06-2021).
- [19] «Use volumes,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/storage/volumes/> (visitado 04-06-2021).
- [20] «Use bind mounts,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/storage/bind-mounts/> (visitado 04-06-2021).
- [21] «Networking overview,» Docker Documentation. (2 de jun. de 2021), dirección: <https://docs.docker.com/network/> (visitado 04-06-2021).
- [22] *Docker (software)*, en *Wikipedia*, 6 de jun. de 2021. dirección: [https://en.wikipedia.org/w/index.php?title=Docker_\(software\)&oldid=1027143347](https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1027143347) (visitado 09-06-2021).
- [23] «Dockerfile reference,» Docker Documentation. (9 de jun. de 2021), dirección: <https://docs.docker.com/engine/reference/builder/> (visitado 09-06-2021).
- [24] H. Jethva. «How Dockerfile Works? – Linux Hint.()», dirección: https://linuxhint.com/dockerfile_beginner_guide/ (visitado 10-06-2021).
- [25] «Best practices for writing Dockerfiles,» Docker Documentation. (9 de jun. de 2021), dirección: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (visitado 10-06-2021).
- [26] T. Donohue. «How To Communicate Between Docker Containers,» Tutorial Works. (6 de nov. de 2020), dirección: <https://www.tutorialworks.com/container-networking/> (visitado 10-06-2021).
- [27] «Docker network connect,» Docker Documentation. (9 de jun. de 2021), dirección: https://docs.docker.com/engine/reference/commandline/network_connect/ (visitado 10-06-2021).
- [28] «Legacy container links,» Docker Documentation. (9 de jun. de 2021), dirección: <https://docs.docker.com/network/links/> (visitado 10-06-2021).
- [29] «Overview of Docker Compose,» Docker Documentation. (11 de jun. de 2021), dirección: <https://docs.docker.com/compose/> (visitado 12-06-2021).
- [30] «Docker Compose Tutorial: Advanced Docker made simple,» Eduative: Interactive Courses for Software Developers. (), dirección: <https://www.educative.io/blog/docker-compose-tutorial> (visitado 12-06-2021).
- [31] «Cómo instalar WordPress con Docker Compose,» DigitalOcean. (), dirección: <https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-with-docker-compose-es> (visitado 12-06-2021).
- [32] «¿Qué es la organización en contenedores?» (), dirección: <https://www.redhat.com/es/topics/containers/what-is-container-orchestration> (visitado 13-06-2021).
- [33] «Container Orchestration,» VMware. (), dirección: <https://www.vmware.com/topics/glossary/content/container-orchestration> (visitado 13-06-2021).
- [34] «Archivo:Kubernetes logo.svg - Wikipedia, la enciclopedia libre.()», dirección: https://commons.wikimedia.org/wiki/File:Kubernetes_logo.svg (visitado 13-06-2021).
- [35] «What is Kubernetes?» Kubernetes. (), dirección: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> (visitado 13-06-2021).

- [36] «Kubernetes Components,» Kubernetes. (), dirección: <https://kubernetes.io/docs/concepts/overview/components/> (visitado 13-06-2021).
- [37] «Use a Service to Access an Application in a Cluster,» Kubernetes. (), dirección: <https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/> (visitado 13-06-2021).
- [38] «Swarm.» (), dirección: https://hub.docker.com/_/swarm (visitado 13-06-2021).
- [39] *Docker/Classicswarm*, Docker, 13 de jun. de 2021. dirección: <https://github.com/docker/classicswarm> (visitado 13-06-2021).
- [40] *Docker/Swarmkit*, Docker, 13 de jun. de 2021. dirección: <https://github.com/docker/swarmkit> (visitado 13-06-2021).
- [41] «Swarm mode overview,» Docker Documentation. (11 de jun. de 2021), dirección: <https://docs.docker.com/engine/swarm/> (visitado 13-06-2021).
- [42] «In-Depth Explanation of Swarm - Programmer Sought.» (), dirección: <https://www.programmersought.com/article/9158256489/> (visitado 13-06-2021).
- [43] «Swarm mode key concepts,» Docker Documentation. (11 de jun. de 2021), dirección: <https://docs.docker.com/engine/swarm/key-concepts/> (visitado 13-06-2021).
- [44] «Don't Panic: Kubernetes and Docker,» Kubernetes. (2 de dic. de 2020), dirección: <https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/> (visitado 13-06-2021).
- [45] *Cgroups*, en Wikipedia, 10 de jun. de 2021. dirección: <https://en.wikipedia.org/w/index.php?title=Cgroups&oldid=1027879091> (visitado 13-06-2021).
- [46] *Linux namespaces*, en Wikipedia, 1 de jun. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Linux_namespaces&oldid=1026275774 (visitado 13-06-2021).
- [47] «Docker security,» Docker Documentation. (9 de jun. de 2021), dirección: <https://docs.docker.com/engine/security/> (visitado 09-06-2021).
- [48] *OpenVZ*, en Wikipedia, 7 de ene. de 2021. dirección: <https://en.wikipedia.org/w/index.php?title=OpenVZ&oldid=998912714> (visitado 14-06-2021).
- [49] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu y T. Jaeger, «Security Namespace: Making Linux Security Frameworks Available to Containers,» en *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, ago. de 2018, págs. 1423-1439, ISBN: 978-1-939133-04-5. dirección: <https://www.usenix.org/conference/usenixsecurity18/presentation/sun>.
- [50] «Capabilities(7) - Linux Manual Page.» (), dirección: <https://man7.org/linux/man-pages/man7/capabilities.7.html> (visitado 14-06-2021).
- [51] N. T. Blog. «Evolving Container Security With Linux User Namespaces,» Medium. (7 de ene. de 2021), dirección: <https://netflixtechblog.com/evolving-container-security-with-linux-user-namespaces-afbe3308c082> (visitado 14-06-2021).
- [52] «Documentation for /Proc/Sys/Fs/ — The Linux Kernel Documentation.» (), dirección: <https://www.kernel.org/doc/html/latest/admin-guide/sysctl/fs.html#overflowgid-overflowuid> (visitado 14-06-2021).
- [53] «Content trust in Docker,» Docker Documentation. (11 de jun. de 2021), dirección: <https://docs.docker.com/engine/security/trust/> (visitado 14-06-2021).

- [54] «Grsecurity.» (), dirección: <https://grsecurity.net/> (visitado 14-06-2021).
- [55] *PaX*, en *Wikipedia, la enciclopedia libre*, 22 de jul. de 2019. dirección: <https://es.wikipedia.org/w/index.php?title=PaX&oldid=117622364> (visitado 14-06-2021).
- [56] «Docker and iptables,» Docker Documentation. (11 de jun. de 2021), dirección: <https://docs.docker.com/network/iptables/> (visitado 14-06-2021).
- [57] T. Bui. «Analysis of Docker Security.» arXiv: 1501.02967 [cs]. (13 de ene. de 2015), dirección: <http://arxiv.org/abs/1501.02967> (visitado 14-06-2021).