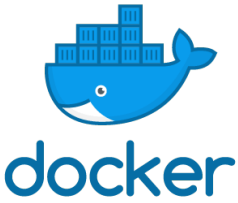
UNIVERSIDAD
POLITÉCNICA
DE MADRID



Análisis de contenedores Docker

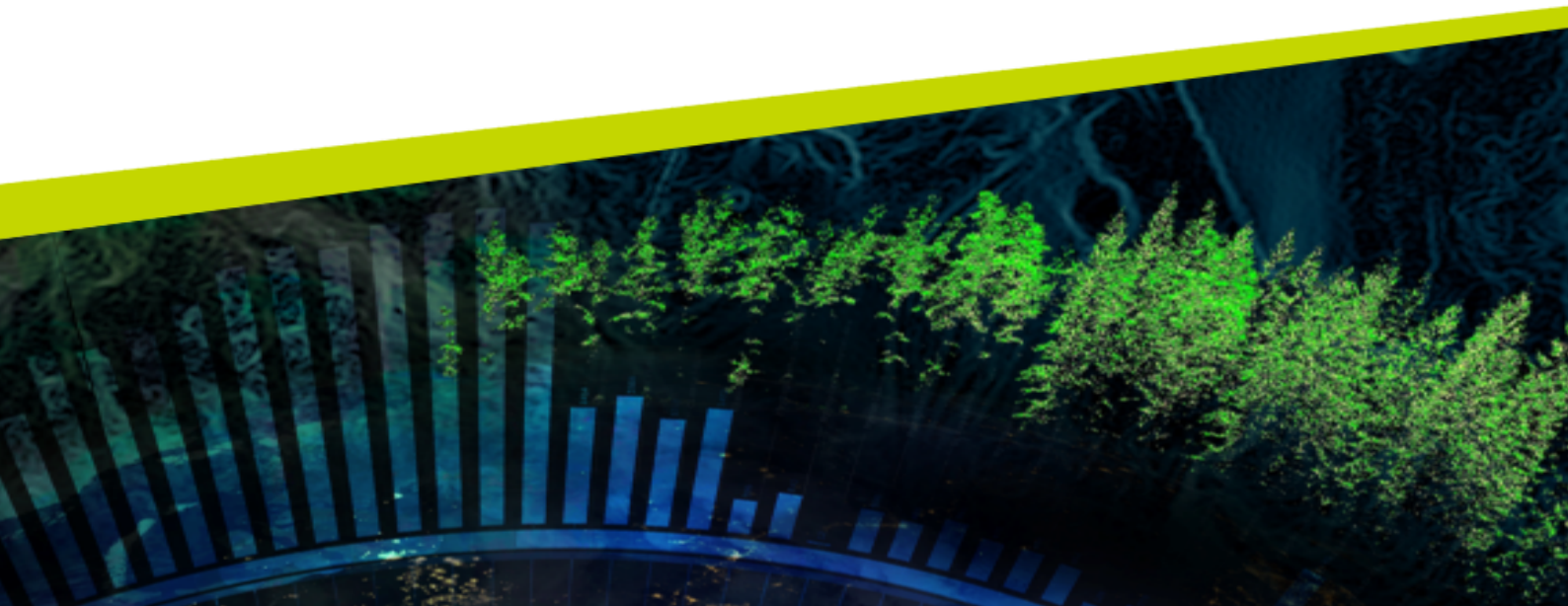
- y sus implicaciones de seguridad

Javier Alonso Silva

Seguridad en Sistemas y Redes

Universidad Politécnica de Madrid

2021



Resumen

TO–DO

Índice

1. Introducción

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):

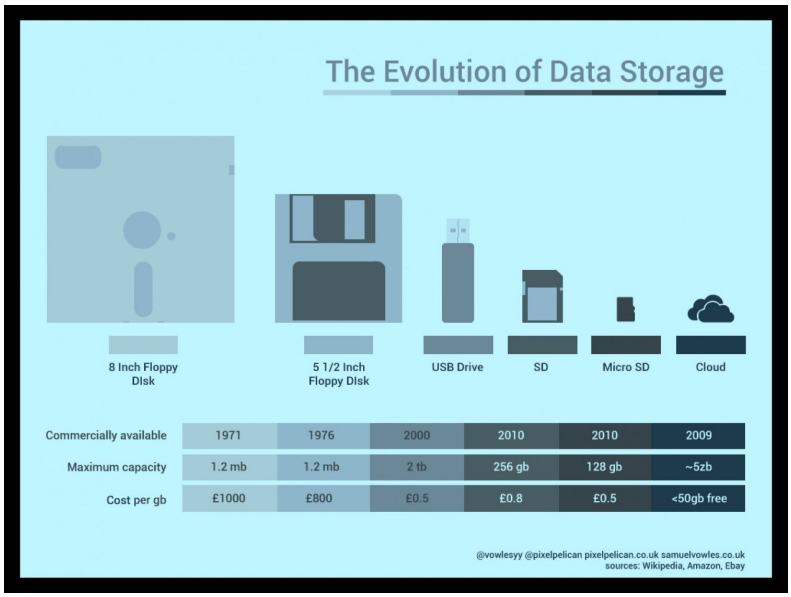


Figura 2: Evolución del espacio de almacenamiento en términos económicos y cuantitativos [3].

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).

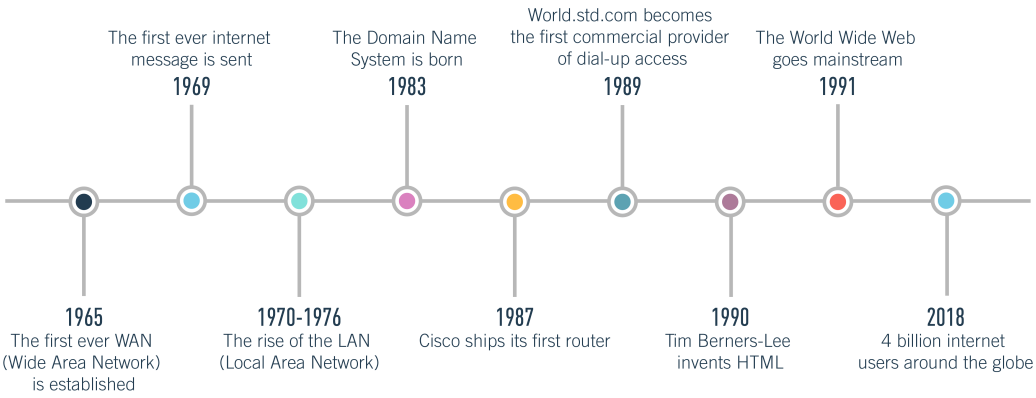


Figura 3: Evolución de Internet a lo largo del tiempo, hasta llegar a hoy [4].

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, librerías 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):

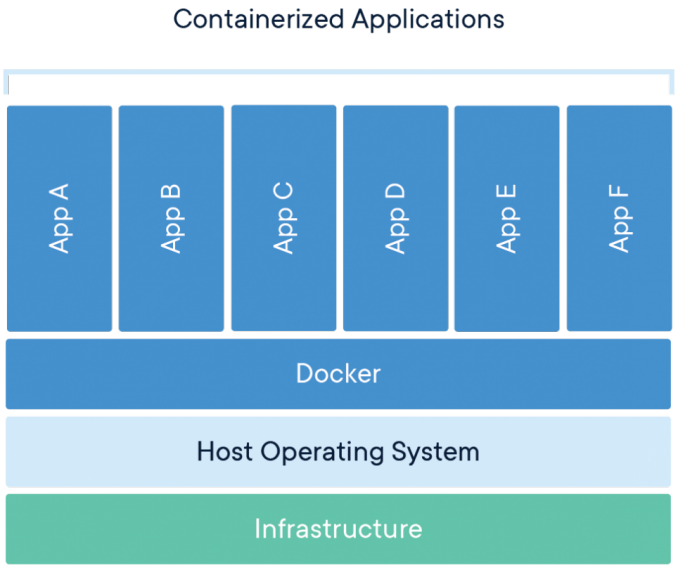


Figura 4: Distribución de los contenedores sobre el motor de ejecución de Docker [7].

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.

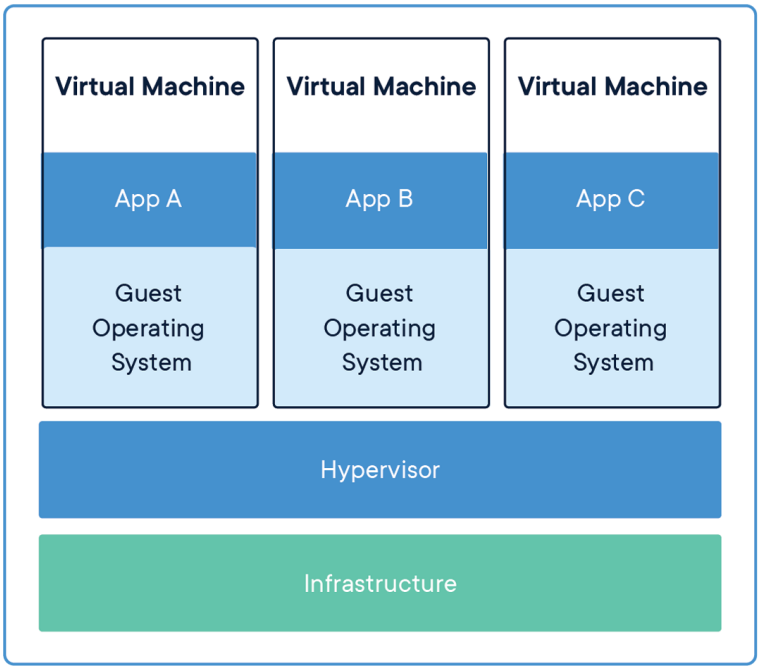


Figura 5: Capas de abstracción de una máquina virtual sobre una máquina anfitriona [7].

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*

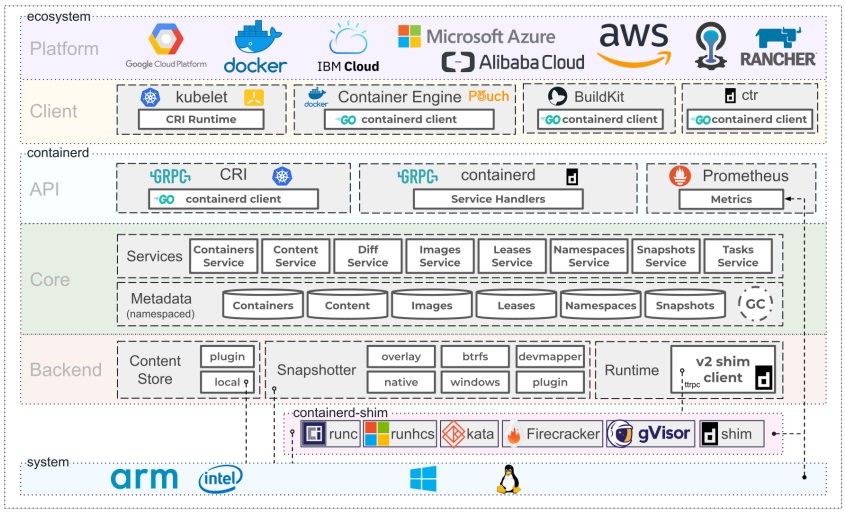


Figura 6: Entorno de ejecución de *containerd* basado en `runC` de la OCI [8].

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].

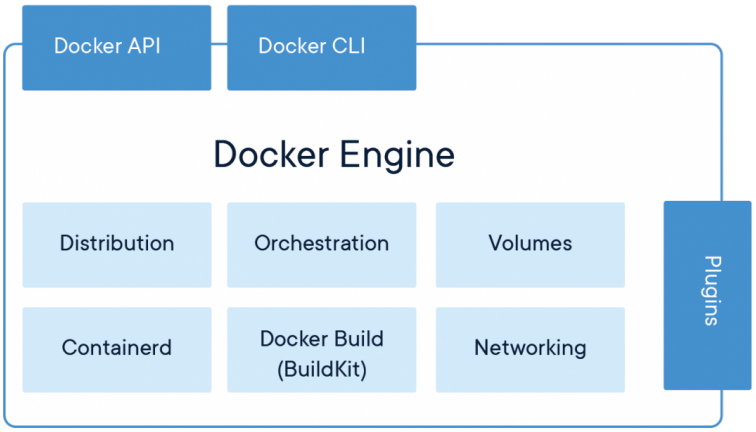


Figura 7: Arquitectura del entorno de ejecución de Docker [9].

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 distribuídas.
- *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 $1 \ldots n$ 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 deshuso 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:

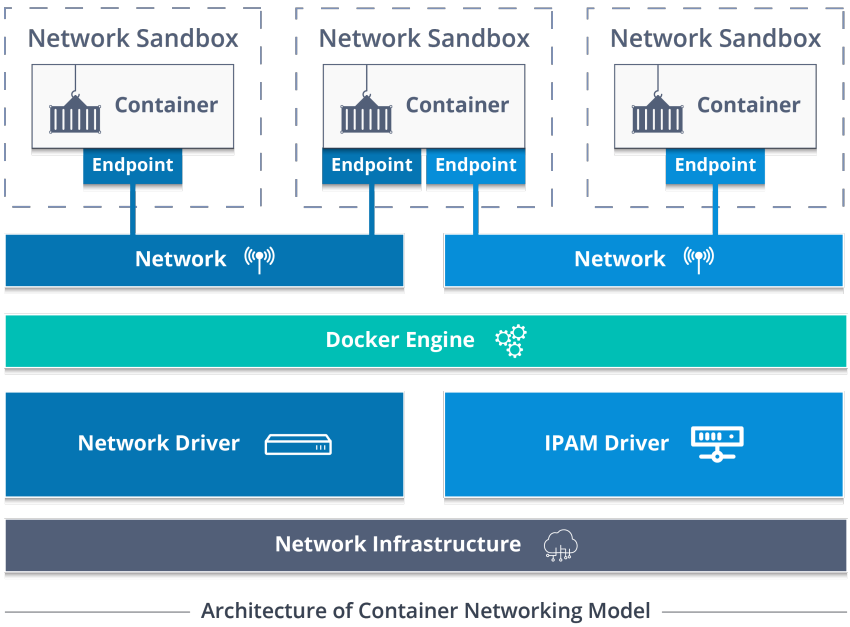


Figura 8: Comunicación entre contenedores usando el motor de Docker [12].

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*

2. Docker

2.1. Estructura de un Docker

2.2. Creación de un contenedor

2.3. Comunicación entre contenedores

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

2.5. "Orquestación" de contenedores

2.6. Líneas futuras de desarrollo e innovación

3. Seguridad en Docker

3.1. Análisis de la pila Docker

3.2. Diferencias fundamentales con `chroot`

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

3.4. Seguridad en las comunicaciones inter–contenedores

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).