UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

Una aproximación «down-top» a los contenedores

Autor:
Ignacio Castelo González

Tutor:
Tomás Fernández Pena

Grao en Enxeñaría Informática

Junio 2021

Traballo de Fin de Grao presentado na Escola Técnica Superior de Enxeñaría da Universidade de Santiago de Compostela para a obtención do Grao en Enxeñaría Informática

Resumen

Una gran parte de los usuarios habituales de contenedores trabajan con esta tecnología abstrayéndose de ciertos aspectos sobre los que se fundamenta. Esto también ocurre con los usuarios noveles.

Con este trabajo se pretende crear una guía introductoria a los conceptos básicos que forman los contenedores, analizando varios niveles de abstracción. En concreto, se propone realizar un estudio comenzando por las características que ofrece el *Linux Kernel*, subiendo los niveles, y pasando por *Container Runtimes*, *Container Engines* y *Orchestrators*. Además, también se explicarán otros conceptos relacionados como imágenes y estándares que se utilizan.

Todo esto se hará basándose, principalmente, en la arquitectura de *Docker* y *Kubernetes* ya que son hoy en día las dos tecnologías más conocidas y usadas en lo que a contenedores se refiere.

Finalmente, se incluirán algunos apartados que podrán ser de ayuda para una mayor comprensión de la materia, explicando la historia de los contenedores y el estado en el que se encuentran actualmente.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introdución

1.1. Motivación del proyecto

Durante los últimos años, varios ámbitos de la informática, entre los que podemos encontrar, principalmente, microservicios o la práctica de *DevOps*, han comenzado a utilizar los contenedores como una de sus herramientas fundamentales.

Normalmente, cuando un ingeniero informático o programador se introduce en estos temas, no le presta demasiada atención a los contenedores en sí, sino que los trata como una simple herramienta que presenta unos resultados, abstrayendo los detalles de su funcionamiento.

Además, si un usuario intenta buscar información en la red o en los libros sobre cómo funcionan, no le será una tarea fácil obtener la visión global que está buscando. Esto sucede porque no hay mucha información en esas fuentes. Y la que tiene utilidad, se encuentra con bastante dificultad.

Por lo tanto, en este trabajo se recopilará y presentará de forma clara, concisa y ordenada la información necesaria para que los usuarios que lo deseen puedan consultarla.

Los datos que se trabajarán en el informe se presentan con más detalle en los objetivos del trabajo.

1.2. Objetivos

- Introducir al usuario en los principales conceptos relacionados con la virtualización, comparando las máquinas virtuales con los contenedores.
- Explicar la historia de los contenedores y describir algunas alternativas que existen actualmente.
- Analizar cada una de las partes que componen un contenedor partiendo del *Stack* de Docker.

- Presentar los anteriores puntos en una página web de una forma menos cruda y atractiva añadiendo un tutorial sobre cómo crear un contenedor simple desde cero en Bash y en Go.

1.2.1. Consideraciones

Dada la naturaleza del proyecto no se requiere un estudio referente a cómo afrontarlo desde el punto de vista comercial, económico, técnico o legal.

El trabajo se basa, simplemente, en la recopilación y procesamiento de información que se encuentra en la Internet, es decir, es un estudio técnico. El uso de la información no implica ninguna operación en la que haya que desarrollar un sistema por lo que no conlleva a la creación de un producto comercial o que precise de una inversión. Por tanto, el estudio no requerirá ningún tipo de análisis de viabilidad.

1.3. Mecanismos de virtualización

1.3.1. Virtualización

La virtualización [1] es un proceso mediante el cual un software es usado para crear una abstracción sobre los recursos que queremos virtualizar. Esto da lugar a una percepción en la que los elementos hardware de la computadora física se dividen en varias computadoras virtuales.

Existen dos principales tipos de técnicas de virtualización:

- Basada en máquinas virtuales (VM).
- Basada en contenedores (también denominada virtualización ligera o a nivel de sistema operativo).

1.3.2. Virtualización mediante VM

Este paradigma virtualiza el sistema operativo completo. Le presenta abstracciones a la máquina virtualizada de los discos virtuales, CPUs virtuales, etc. Se podría resumir en que está virtualizando el ISA (*Instruction Set Architecture*) completo.

Con esta técnica varios sistemas operativos pueden compartir los mismos recursos hardware (realmente comparten una representación virtual de estos recursos). Así, las máquinas *guest* podrán hacer operaciones de lectura y escritura sobre un disco duro virtual creyendo que es un disco duro físico y que sólo ellas tienen acceso [1].

Para poder virtualizar más sistemas operativos aparte del *host* se necesita un software llamado *hypervisor* [2, 3].

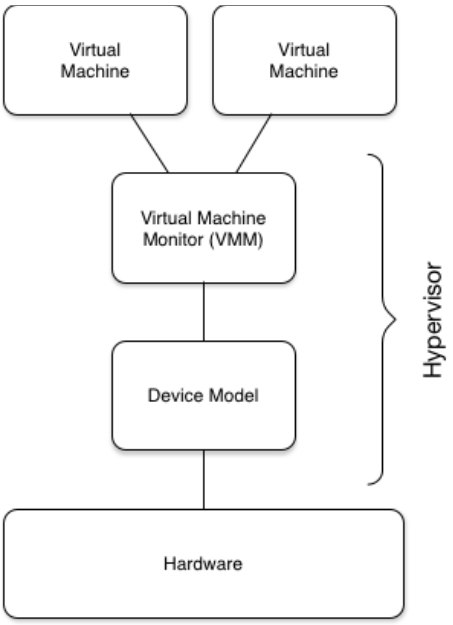

Figura 1.1: Arquitectura del *hypervisor*. Fuente: `https://medium.com/@penberg`

1.3.3. Hypervisor

El *hypervisor* [1, 4] es la pieza especial de software encargada de permitir la virtualización de varios sistemas operativos. Proporciona aislamiento entre máquinas virtuales, que funcionan de forma independiente y pueden tener cada una su propio sistema operativo. Está formado de dos partes, tal y como se muestra en la figura 1.1.

- *Virtual Machine Monitor* (VMM): es el encargado de manejar las *traps* causadas por el sistema operativo del *guest* para ejecutar instrucciones privilegiadas como el acceso a E/S. Esta pieza fundamental debe cumplir tres propiedades:
  - **Aislamiento y control de recursos:** debe aislar todos los *guests* entre sí y manejar la división de recursos hardware que se le asignan a cada uno.
  - **Equivalencia**: el *guest* debe tener igual comportamiento que el que tendría ejecutándose directamente en hardware.
  - **Rendimiento**: el sobrecoste del *hypervisor* debe ser el menor posible. Para esto, deberían ejecutarse la mayor parte de las instrucciones directamente en los dispositivos físicos sin necesidad de ser traducidas.
- *Device model*: es el encargado de la virtualización de E/S implementando las interfaces necesarias para manejar las *traps* y emularlas, para luego devolver las interrupciones a la máquina virtual correspondiente.

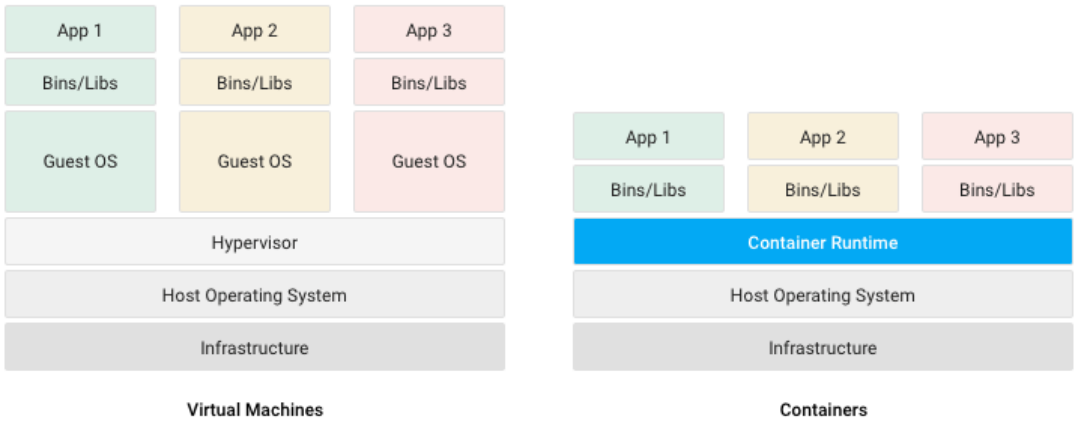

Figura 1.2: Diferencia VM y Contenedores. Fuente: `https://cloud.google.com/containers`

Una de las mayores críticas sobre el uso del *hypervisor* es que suele ser muy pesado comparado con otras técnicas.

1.3.4. Virtualización mediante contenedores

Este método no crea una abstracción del hardware sino que usa directamente las herramientas que proporciona el Linux Kernel (como *cgroups* y *namespaces*) para aislar los distintos recursos. Crea un límite lógico dentro del propio sistema operativo [1].

Por lo tanto, un contenedor es una forma de virtualización de sistema operativo ligera y ágil. En vez de desplegar una máquina virtual completa, un sólo contenedor puede ser usado para ejecutar un proceso, microservicio o incluso una aplicación monolítica más grande. Dentro del contenedor se encuentran las partes necesarias para la correcta ejecución del programa: ejecutables, código binario, librerías y archivos de configuración. Esto permite que puedan funcionar en prácticamente cualquier computadora, desde un portátil hasta un servidor en la nube. Gracias a que utilizan las herramientas del Linux Kernel no necesitan un nuevo sistema operativo para cada contenedor, ya que pueden compartir todos el sistema operativo del *host* [5, 6, 7]. Las principales diferencias en cuanto a la infraestructura de los contenedores y las máquinas virtuales se pueden observar en la figura 1.2.

Uso de los contenedores

El uso de los contenedores está creciendo exponencialmente durante los últimos años, principalmente en el ámbito de computación en la nube. Además, existen usuarios y empresas que utilizan los contenedores como plataforma de

virtualización principal en sustitución de las máquinas virtuales [8]. Los ámbitos en los que se pueden exprimir realmente sus ventajas son:

- **Aplicaciones independientes** [10]: los contenedores permiten abstraerse del entorno en el que se ejecutan las aplicaciones sin preocuparse por detalles como las versiones de software específicas o configuraciones.
- **Microservicios** [8]: las arquitecturas de microservicios consisten en aplicaciones compuestas por muchos servicios pequeños poco acoplados e independientes entre sí. De esta forma, los contenedores al ser pequeños y ligeros resultan una muy buena opción para usar en este ámbito.
- ***DevOps* aplicado a *CI/CD*** [6]: la tecnología de contenedores facilita el ciclo «*build, test and deploy*» sobre las mismas imágenes.
- ***Multi-Cloud* e *Hybrid-Cloud*** [8]: se aprovechan de que los contenedores pueden funcionar de forma consistente en prácticamente cualquier lugar.
- **Modernización y migración** [8]: una de las técnicas más comunes para la modernización de aplicaciones es encapsularlas en contenedores para que puedan ser migradas a la nube.

Ventajas de los contenedores

Las principales ventajas de los contenedores son las siguientes [13]

- **Rapidez y ligereza**: al tener una arquitectura en la que el *kernel* está compartido, la velocidad a la que se inician es extremadamente alta, poco más que lanzar un proceso corriente.
- **Gestión**: existen actualmente un gran número de herramientas que ayudan a monitorizar, configurar y desplegar contenedores.
- ***Disk footprint***[1]: Gracias al uso de *Union Filesystems* y a la técnica de *Copy on Write* que estudiaremos más adelante se puede ahorrar en gran medida la cantidad de disco que usan los contenedores. Además, al reducir las aplicaciones a, simplemente, el código, librerías, componentes e interfaces, también se ahorra disco en comparación con otras técnicas de virtualización.
- **Portabilidad e independencia de plataformas** [8]: al llevar todas las dependencias de los contenedores con ellos, el software se puede escribir en un entorno y trasladar a otro sin necesidad de ser reconfigurado.

[1]Es el tamaño que ocupan los datos del contenedor en el disco cuando están en un estado inactivo [11]

- **Habilidad para congelar y descongelar contenedores**: mediante una señal `SIGSTOP` a todos los procesos que forman un contenedor (mediante los grupos de control que veremos más adelante) se puede ahorrar energía de forma masiva tanto en un *datacenter* como en un ordenador personal.

- **Facilidad de migración**: en otros tipos de virtualización como en el caso de las máquinas virtuales la migración entre *hosts* puele ser muy costosa, no sólo por su gran tamaño, sino porque puede haber incompatibilidades en formatos de archivo e imágenes. En el caso de los contenedores es mucho más sencillo si se comparte el mismo *kernel* entre *hosts*. Cabe destacar que se están diseñando herramientas que pueden permitir migrar contenedores sin que se dejen de ejecutar.

- **Escalabilidad** [7, 12]: como unidad de escalabilidad, la máquina virtual es cara. Los contenedores pueden ser creados y destruidos fácilmente y con muy pocos recursos.

Desventajas de los contenedores

Algunas de las contras más importantes son las siguientes:

- **Seguridad** [14]: el aislamiento entre contenedores y entre estos y el *host* es menor que en el caso de las máquinas virtuales, lo que puede provocar vulnerabilidades en el *host*. Además, comparándolos con las máquinas virtuales, estas últimas podrían recomponerse mediante las *snapshots* realizadas anteriormente en caso de un ataque. Finalmente, es necesario puntualizar que, al estar formados por muchas capas, todas ellas deberían ser protegidas [15].

- **Monitorización** [16]: la infraestructura está formada por varias capas las cuales hay que controlar y gestionar, aunque únicamente se tenga una sola aplicación en un contenedor.

- **Complejidad a gran escala** [13]: muchos de los *frameworks* de orquestación no implementan todas las medidas de seguridad adecuadas o no presentan ciertas funcionalidades importantes. Es uno de los grandes retos que supone escalar un sistema basado en contenedores.

1.4. Organización del documento

El resto de la memoria se divide en tres partes. El capítulo 2 describe cómo han ido evolucionando a los largo de los años las tecnologías de contenedores, además de realizar un pequeño estudio de las diferentes alternativas que podemos encontrar en la actualidad. En el capítulo 3 se explican con detalle las diferentes capas y herramientas que componen las tecnologías que se utilizan para crear y gestionar

contenedores. Por último, en el capítulo 4 se incluyen algunas recomendaciones para ampliar los conocimientos sobre la materia.

Capítulo 2

Historia y tecnologías actuales

2.1. Historia de los contenedores

Los primeros orígenes del término «multiprogramación» (idea seminal en la que se basan los contenedores y otros sistemas de virtualización) datan de la década de 1950. Con esta nueva característica se incrementó la complejidad del software de los sistemas (dado que permitía a varios procesos simultáneos interactuar entre ellos y compartir recursos). A raíz de este suceso comenzaron a surgir las primeras máquinas virtuales con el objetivo de compartir de forma segura los recursos de una máquina física entre varios procesos simultáneos.

2.1.1. Chroot (1979)

No se comenzó a trabajar en conceptos propios de contenedores hasta la séptima edición de UNIX lanzada por Bell Labs en 1979 [17]. En este año se introdujo en el *kernel* la llamada al sistema `chroot`[1] que permitía cambiar el directorio raíz de un proceso y de sus hijos a un nuevo lugar dentro del sistema de archivos. De esta forma se consigue que un grupo de procesos tengan una visión del almacenamiento del sistema reducida a unos archivos y directorios específicos [20]. Este avance fue el comienzo del aislamiento de los procesos [25]. El problema principal era que los procesos con permisos de root podían saltarse fácilmente el aislamiento [21].

2.1.2. FreeBSD Jails (2000)

En el año 2000 se creó el **primer** «**pseudo-contenedor**» que realmente sería el impulsor de las tecnologías de contenedores que existen hoy en día, las Jails de FreeBSD.

[1] El Dr. Marshall Kirk Mckusick deduce que el motivo por el que fue añadido consistía en poder hacer `chroot` dentro del árbol de directorios /4.2BSD y construir el sistema usando únicamente los archivos necesarios dentro de ese mismo directorio [18].

El problema que se buscaba resolver era «confinar el root omnipotente», que da nombre al documento original donde se presenta esta herramienta [18].

Las Jails dan uso a `chroot` añadiendo nuevos mecanismos existentes en el sistema operativo dando lugar a un entorno «equivalente» a una máquina virtual. En ella los procesos y sus hijos pueden manipular únicamente los servicios y archivos a los que se le da acceso. De ahí que se le de el nombre de *jail* (cárcel en inglés), ya que el objetivo principal es proporcionar seguridad como en las celdas de una prisión.

La aproximación seguida por las Jails consiste en mantener el modelo de seguridad de UNIX permitiendo el root omnipotente pero confinándolo en una celda o *jail* limitando su alcance a la propia celda. Por tanto, el administrador del sistema puede separar el sistema en varias celdas asignando a cada una un superusuario sin perder el control del sistema completo.

Introducir un proceso en una *jail* es un viaje sólo de ida; tanto el proceso como sus descendientes se quedarán en la *jail* y no podrán salir de ninguna manera.

Aunque habíamos comentado que existen formas de salir del `chroot`, esta tecnología bloquea los mecanismos tradicionales que utilizaban los *crackers*[2].

Alguna característica de red digna de mencionar es que, en un principio, cada celda tenía asociada una única dirección IPv4 (única y no compartida con el resto de celdas), aunque, posteriormente, se implementó una opción que permite tener más. Además, para mantener la seguridad deseada (por ejemplo, para evitar *spoofing*) se desactivaron ciertas llamadas al sistema, como la posibilidad de crear *raw sockets*, lo que impide utilizar algunas instrucciones tan comunes como `ping`.

Otra característica es que para aumentar la eficiencia de almacenamiento se eliminaron ciertos binarios y archivos «poco relevantes»[3] de la celda.

Por último, cabe destacar que, respecto a los usuarios, cada celda tiene sus propios UIDs y GIDs. Un mismo usuario en una celda puede corresponderse con otro usuario distinto en otra celda (aunque esto sólo se podría comprobar desde el *host*) [26, 18].

2.1.3. Linux VServer (2001)

Más tarde, en el año 2001, se añadió al Linux Kernel otro mecanismo de celdas en el que se incluye control de recursos y aislamiento en el sistema de archivos, direcciones de red y memoria [17, 25]. Estas funcionalidades permitieron crear particiones llamadas «contextos de seguridad». El sistema virtualizado dentro de la partición funciona como un VPS (*Virtual Private Server*[4]) [22, 21, 28].

[2]Curiosamente, los enlaces duros siguen funcionando, lo que puede traducirse en una brecha de seguridad [26, 18]

[3]Dadas estas imprecisiones en la forma de redactar el documento da la sensación de que los creadores no se imaginaban hasta dónde iban a trascender las *jails* [18].

[4]Un VPS es una partición que se hace sobre un servidor físico y que obtiene recursos dedicados [150].

2.1.4. Solaris Zones (2004)

A medida que avanzaban los años se demandaba cada vez más la posibilidad de tener múltiples trabajos en un mismo sistema. Es decir, no sólo se buscaba mantener la seguridad, sino también reducir costes y mejorar el uso de recursos a gran escala [17, 27]

En Sun Microsystems propusieron una solución en 2004: los Solaris Containers o Zones, que permiten aislar procesos formando grupos.

El objetivo era lograr que los administradores necesitaran unos pocos minutos para configurar y lanzar una nueva *zone*. El sistema debería crearla automáticamente utilizando el *mount namespace*[5] y añadiendo límites en los recursos compartidos que se consumían (inicialmente sólo para CPU).

El administrador, mediante la línea de comandos, puede configurar las propiedades de las *zones* en tiempo real o mediante un script. En dichas propiedades se incluye información del sistema de archivos, direcciones IPv4, dispositivos y límites en la asignación de recursos. Se pretendía dar soporte a aplicaciones comerciales que puedan ser potencialmente escalables y fuertemente dependientes de la red.

El sistema operativo identifica dos tipos de zonas:

- Global Zone: es el *zone* por defecto y tiene control sobre todos los procesos. Siempre existe aunque el usuario no haya creado ninguna manualmente.
- Non-global Zone: son las *zones* configuradas dentro de la global. Los procesos de una no afectan a los procesos de otra ni se pueden ver. La forma de aislarlas del sistema físico es mediante una capa, la plataforma virtual.

Algo a tener en cuenta es que las Zones se preocupan por mantener ciertas utilidades sin dejar a un lado la seguridad; por ejemplo, no se permite crear *raw sockets* pero dan acceso al protocolo TCP para permitir programas como `ping` [26, 17]. Además, se permite virtualizar otras versiones de Solaris y Linux gracias a su sistema de traducción de llamadas al sistema por *zone* [19].

2.1.5. OpenVZ (previamente Virtuozzo) (2005)

Un año más tarde, en 2005, Virtuozzo, mediante un parche, consiguió implementar en el Linux Kernel nuevos límites en el uso de los recursos, el aislamiento de sistemas de archivos, usuarios, IPC y dispositivos [17]. Este parche nunca fue añadido a la rama principal de Linux.

Una gran desventaja de OpenVZ es que no sólo es necesario tener una rama no oficial de Linux, sino que todos sus «pseudo-contenedores» deben compartir la misma [21].

[5]El *mount namespace, incluido en el kernel de Linux en el año 2002, permite aislar ciertas partes del sistema de archivos. En el capítulo 3.1.1 trataremos en detalle este y otros namespaces.*

2.1.6. Process Containers (cgroups) y Namespaces (2006-07)

En el año 2006, el trabajador de Google, Paul Menage, propuso un nuevo mecanismo generalizado para agrupar procesos gracias a un *framework* que aprovecha los mecanismos de control existentes. El objetivo era que los usuarios se puedan centrar en el controlador de recursos y se abstraigan de cómo los procesos son monitorizados y gestionados [29].

Dicho *framework* fue renombrado posteriormente a «*control groups*» (o grupos de control). Esta funcionalidad fue uno de los hitos que impulsó los contenedores hasta convertirlos en lo que son actualmente [24].

Por otro lado, desde 2002 ya existía un *namespace* (el ya nombrado *mount namespace*). Sin embargo, fue a partir de 2007 cuando se comenzó a ver el potencial que podían tener junto con otras funcionalidades como los grupos de control para crear contenedores. De aquí nació el siguiente *namespace* (el *network namespace*) [30, 79], hasta hoy, que ya existen 8 tipos distintos: *cgroup, IPC, network, mount, PID, time, user* y *UTS* [31].

2.1.7. LXC (2008)

LXC (Linux Containers) se podrían considerar los **primeros contenedores** tal y como conocemos el concepto hoy en día, ya que fue la primera tecnología en aplicar los dos elementos principales de los contenedores: **namespaces** y **cgroups** [25].

Permite al usuario mediante una interfaz en línea de comandos la posibilidad de comunicarse con las facilidades que ofrecen los *namespaces* y *cgroups*. Todo esto en el espacio de usuario.

Sin embargo, no es una herramienta apropiada para realizar tareas de gestión de contenedores en un alto nivel (como la distribución de aplicaciones y servicios) [19].

2.1.8. Warden (2011)

En el año 2011, la plataforma desarrollada por antiguos miembros de VMware, Cloud Foundry [33], inició el proyecto Warden, capaz de aislar múltiples entornos en cualquier sistema operativo. Esta herramienta funciona como un *daemon* y proporciona una API para la gestión de contenedores [32] mediante la arquitectura «cliente-servidor» a múltiples *hosts* [21].

En sus primeras implementaciones utilizaba LXC para gestionar las llamadas al sistema pero más tarde lo sustituyeron por su propia implementación [20]. Hoy en día Cloud Foundry utiliza una re-escritura de Warden llamada Garden [35] (que funciona como un *backend* en su modelo de contenedor para su nueva arquitectura Diego) [34].

2.1.9. LMCTFY (2013)

LMCTFY (or Let Me Containerize That For You) fue la versión open-source del stack de Google a los contenedores en Linux. La intención era proporcionar rendimiento y aprovechar altamente los recursos del sistema sin implicar un gran sobrecoste [20]. Además, las aplicaciones podían ser escritas siendo «*container-aware*» lo que permite a un contenedor crear sus propios «sub-contenedores» y gestionarlos [23].

El desarrollo finalizó en 2015 y Google comenzó a transferir parte de la implementación a *libcontainer* que ahora es parte de OCI (Open Container Foundation) y herramienta fundamental de Docker [23, 20].

2.1.10. Docker (2013)

Docker supuso un despunte en la popularidad de los contenedores en el año de su salida y el crecimiento de ambos conceptos han ido de la mano desde entonces siendo hoy en día la herramienta líder en este ámbito [32].

En sus inicios utilizaba LXC al igual que Warden, pero más tarde lo sustituyó por su propia librería *libcontainer* [20].

Al ser en la actualidad la tecnología de contenedores más popular, la hemos escogido para ser el ejemplo y el pilar fundamental sobre el que se va a basar el documento.

2.1.11. OCI (2015)

OCI (*Open Container Initiative*) es un proyecto de la Linux Foundation [43] cuyo objetivo es diseñar un estándar abierto para la virtualización a nivel de sistema operativo [41]. Fue establecida en 2015 por Docker y otros líderes de la industria [42].

Después del lanzamiento de Docker, surgió una gran comunidad alrededor de la idea de utilizar los contenedores como un estándar en el desarrollo de servicios. Sin embargo, con el paso de los meses fueron apareciendo nuevos *runtimes*[6] con nuevas funcionalidades y capacidades que satisfacían las necesidades que iban teniendo los equipos de desarrollo, además de nuevas herramientas que complementaban y mejoraban Docker. Este fue el motivo principal por el que surgió la idea de crear un estándar que garantice que cualquier *runtime* pueda correr imágenes producidas por cualquier herramienta [44].

De hecho, Docker donó parte del código al proyecto, en concreto lo que se correspondía con el *container runtime* que usaba: runc.

Actualmente, OCI define dos especificaciones:

[6]Los *runtimes* son las piezas encargadas de gestionar, crear y ejecutar contenedores. En el capítulo 3.2 se tratarán con detalle

- ***Image specification***: define el formato del archivo de las imágenes OCI. El objetivo de esta especificación es permitir una serie de herramientas estandarizadas que puedan preparar, construir y transportar una imágen [45]. A este formato se le llama OCF (*Open Container Format*) [41].
- ***Runtime specification***: define la configuración, entorno de ejecución y ciclo de vida de un contenedor [45].

Más detalles sobre imágenes y *runtimes* se a lo largo del documento, en las secciones 3.2 y 3.4

2.2. Tecnologías actuales

2.2.1. Docker

Originalmente su objetivo era extender las funcionalidades que permitía LXC. Hoy en día es la plataforma líder en el mercado [36]. Seguramente la popularidad de Docker se debe a cuatro razones:

- **Rápido despliegue**: Docker permite realizar una integración y construcción muy rápida, además de poder reproducir fácilmente un entorno de un contenedor y testearlo [38]. Además, incluye un CLI atractivo para los usuarios comparado con las alternativas que existían en su lanzamiento.
- **Código libre**: por el año 2013 fue precisamente cuando el código libre comenzó a ser el estándar de producción de software [39].
- **Lanzamiento en el momento preciso**: aproximadamente en el 2013 fue cuando los contenedores realmente empezaron a ganar terreno a las máquinas virtuales, en parte, gracias a la popularidad de la filosofía *DevOps* que surgió por el año 2010 (ya que utiliza los contenedores como uno de sus pilares fundamentales) [39].
- **Docker Hub**: es un servicio que permite buscar y compartir imágenes de forma pública como privada (o incluso con un equipo de desarrollo) [40]. En él existen más de 100.000 imágenes con diversas aplicaciones de terceros ya configuradas y listas para su despliegue [36].

Por otro lado, Docker cuenta con una arquitectura basada en cliente-servidor donde el cliente habla a un *Docker daemon* (que puede estar en el cliente o en un servidor remoto) utilizando una API REST. Este componente es el que realmente se encarga de construir, ejecutar y distribuir los contenedores [40].

Cabe destacar que cumple los estándares OCI tanto en imagen como en *runtime*.

Destacar, por último, que Docker se puede ejecutar de forma nativa en Linux y Windows [154].

2.2.2. Podman

Podman es un *container engine* que no trabaja con un *daemon*, al igual que Rkt [71] (que ya está obsoleto), y tiene un CLI prácticamente idéntico al de Docker[7]. Es compatible tanto con imágenes como con *container runtimes* OCI [54, 53].

La ventaja principal es precisamente que al no necesitar un *daemon*, no se necesita garantizar ningún tipo de privilegios de root. De esta forma, evitamos que se obtenga acceso a root mediante un ataque [55].

Por otro lado, podemos indicar que este motor permite crear *pods* donde van los contenedores, de forma similar a lo que hace Kubernetes. De esta forma, los contenedores pueden agruparse y ejecutarse unos junto a otros pudiendo comunicarse fácilmente (esto no significa que Podman sea un orquestador como Kubernetes) [56].

Actualmente, Podman no es compatible con Kubernetes, aunque se puede utilizar de forma independiente para gestionar contenedores de de Kubernetes de forma individualizada [57, 62, 63].

2.2.3. Singularity

Cuando entramos en el mundo de HPC (*High Performante Computing[59]*) cambia el paradigma con el que nos enfrentamos a los contenedores: los sistemas HPC utilizan recursos compartidos para ejecutar una gran cantidad de tareas en paralelo. [58].

Singularity permite ejecutar contenedores Docker de forma nativa en los sistemas HPC [151].

Los demás *container engines* que se muestran en este apartado no están creados para eso, ya que sus modelos de seguridad están diseñados para garantizar a los usuarios privilegios en el contenedor (en HPC los usuarios no suelen ser de confianza) [60]. Además, tampoco soportan trabajos batch o aplicaciones basadas en arquitecturas de computación paralela [58].

Por otro lado, Singularity no implementa grupos de control ni ninguna otra funcionalidad de gestión de recursos [61].

Las características más importantes de Singularity se resumirían en las siguientes [58, 151]:

- No hay un *daemon* con permisos de root.
- No hay cambios de contexto ni usuarios, el usuario de un contenedor siempre es el mismo que inició el proceso, por lo que los privilegios no pueden escalar.

[7]En la página web de Podman aluden a que los usuarios que hayan trabajado con Docker pueden utilizar *alias docker=podman* debido a la gran similitud en los comandos de ambas herramientas [54]

- Soporta entornos con recursos compartidos, soporta hardware HPC y presenta alta compatibilidad con GPUs de HPC.

2.2.4. Linux Containers

Los Linux Containers o LXC (ya introducidos en el apartado 2.1.7) crean un nivel de virtualización distinto a las anteriores tecnologías. En este caso se pueden usar tanto para envolver una aplicación específica, como Docker o Podman, o para emular una nueva máquina entera [64]. Debido a esto, los Linux Containers se suelen utilizar como un sustituto de una VM más rápido y ligero [65, 66].

Más tarde se lanzó LXD, que se podría entender como una extensión de LXC: es una API REST que conecta con la librería de LXC, *libxlc*. Esta herramienta crea un *daemon* que aporta más seguridad y permite conexiones mediante HTTP tanto localmente como en remoto. Además, también añade otras características como migración de contenedores o creación de *snapshots* en tiempo real [67].

2.2.5. Otros

OpenVZ

OpenVZ necesita un parche en el kernel para añadir ciertas características, aunque se puede utilizar con cualquier distribución de Linux. Pero las funcionalidades que ofrece son muy parecidas a las de los Linux Containers [69].

Hyper-V Containers

Solución nativa de Windows, aunque no entra en la definición propuesta al principio del documento ya que no se trata de una virtualización a nivel de sistema operativo [70].

Capítulo 3

Arquitectura de los contenedores

En este capítulo se abordará la arquitectura general de los contenedores, desde el nivel más bajo, que corresponde con las instrucciones que se le envían al *kernel* hasta el nivel más alto, que se corresponde con las herramientas orquestadoras.

Dada la ambigüedad con la que muchos profesionales se refieren a cada una de las herramientas que componen esta arquitectura y, dado que dependiendo de la tecnología a usar, pueden cambiar ligeramente las funciones que realizan sus componentes, se usará como referencia el stack de Docker y de Kubernetes.

Por lo tanto, partiendo de que Docker lo componen los elementos representados en la figura 3.1 y que Kubernetes más Docker está compuesto por los elementos de la figura 3.2, podemos establecer todas las partes que vamos a estudiar, plasmadas en la figura 3.3.

Cabe destacar que **cuando nos referimos a Docker existen dos variantes: podemos referirnos al stack completo de Docker o a Docker *daemon* (*dockerd*)**. En este documento cuando decimos Docker nos referimos al stack completo, mientras que el *daemon* será referido de forma explícita

3.1. Componentes del Linux Kernel

En esta sección se discutirán los principales componentes del Linux Kernel que utilizan los contenedores para aislar los procesos y sus recursos. En concreto se han seleccionado cinco características, siendo las más importantes las dos primeras [13]:

- **Namespaces**: proporcionan el aislamiento entre procesos.
- **Control Groups**: implementan la monitorización y limitación de recursos.
- **Union Filesystem** [1]: es un servicio del sistema de archivos que permite unir los contenidos de diferentes sistemas de archivos.

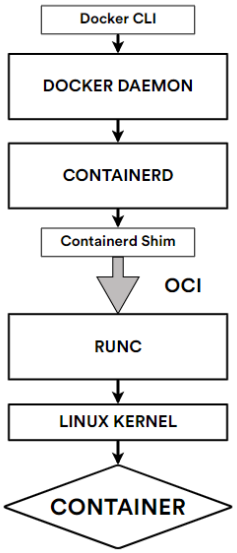

Figura 3.1: Arquitectura Docker

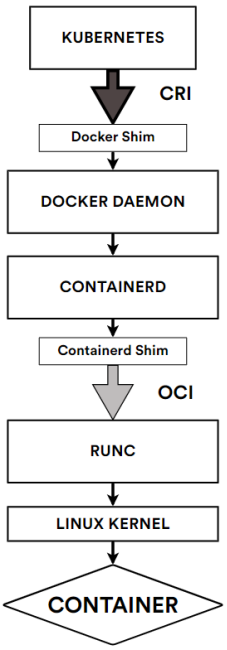

Figura 3.2: Arquitectura Kubernetes con Docker

- **Capabilities** [117]: dividen los poderes del superusuario en distintas unidades que se pueden asociar a hilos.

- **Pivot_root** [73]: mueve la raíz del sistema de archivos del proceso.

3.1.1. Namespaces

En una máquina donde se estén ejecutando varios servicios, es esencial mantener una estabilidad y seguridad. Suponiendo el caso en el que un intruso acceda a uno de los servicios, quizás pueda encontrar una vulnerabilidad en el sistema y acceder a los demás servicios, comprometiendo la máquina completa [77].

El primer *namespace* fue introducido en el *kernel* de Linux en 2002. Desde entonces se han ido implementando paulatinamente hasta siete más. Su función es encapsular ciertos recursos del sistema en una capa de abstracción que hace creer a los procesos dentro de un mismo *namespace* que tienen su propia instancia de un recurso del sistema, aunque realmente lo están compartiendo. De esta forma, se consigue un **aislamiento** entre procesos y aplicaciones dándole la sensación a los mismos de que no hay otros procesos en el sistema [30, 74, 31].

Un *namespace* persiste siempre y cuando tenga procesos asociados. En el momento en el que se disocien todos sus procesos, éste desaparece [81].

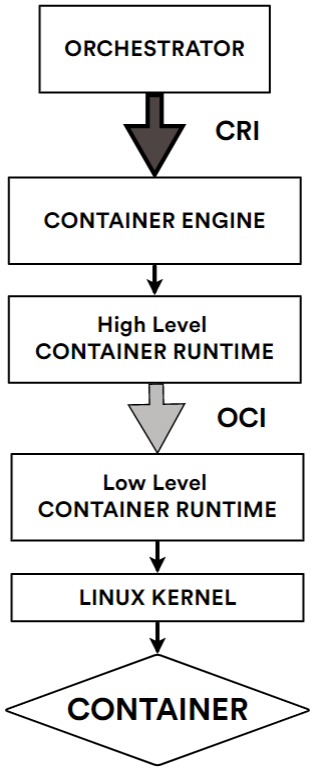

Figura 3.3: Arquitectura de un contenedor a todos los niveles

La API del *kernel* que facilita esta característica consiste de cuatro llamadas principales [31, 74]:

- `clone` [80]: crea un proceso hijo, al igual que la llamada `fork`. Sin embargo, esta llamada proporciona un control más preciso sobre qué partes del contexto de los procesos se comparten y permite añadir ciertas *flags* para que los hijos sean creados en distintos *namespaces*.

- `unshare` [81]: crea un nuevo *namespace* (de uno de los tipos que se comentará más adelante) y, después, ejecuta un proceso dentro de ese *namespace* (si no se especifica ningún proceso, ejecuta por defecto `/bin/sh`). Utiliza los mismos *flags* que clone.

- `setns` [82]: mueve el hilo actual dentro de un *namespace* ya existente.

- `ioctl` [83]: algunas de sus operaciones permiten descubrir información sobre *namespaces*.

Cierta información sobre los *namespaces* está incluida en el *pseudo-filesystem*

`/proc`. En concreto, el `/proc/$PID/ns` donde cada archivo es un *magic link* [1] con el que maneja la información.

En la actualidad existen ocho tipos distintos de *namespaces*. La tarea que realiza cada uno es encapsular un recurso particular del sistema:

Mount NS

Fue el primer *namespace* implementado en 2002, en la versión 2.4.19 de Linux. En aquel momento no se conocía ni la importancia que iba a tener en la actualidad ni que iban a implementarse otros nuevos, por esto la *flag* correspondiente en los comandos de `clone` o `unshare` es `CLONE_NEWNS` [74].

Para entender este *namespace* es necesario saber que Linux mantiene una estructura de datos llamada `vfsmount` para gestionar todos los puntos de montaje del sistema [77]. También es importante saber que, además de los *mounts*[2], existen los *bind mounts*[3] que permiten a un directorio (en lugar de un dispositivo como un disco) ser montado en un punto de montaje. Los contenedores trabajan habitualmente con este concepto: cuando se crea un volumen para un contenedor, realmente se está haciendo un *bind mount* de un directorio perteneciente al *host* dentro del sistema de archivos propio del contenedor. Así, se está creando una nueva estructura `vfsmount` en el *namespace* del contenedor manteniéndola aislada del resto del sistema de ficheros del *host* [1].

En resumen, este *namespace* proporciona aislamiento a estas estructuras de datos, de tal forma que los procesos en distinto *namespace* tienen una visión distinta de la jerarquía de los sistemas de archivos [79]. Además, también proporciona más seguridad a otros *namespaces* nuevos como PID, que veremos más adelante.

Algunos de los usos que ofrece son [76]:

- Cada usuario puede tener su propio `/tmp` para aumentar la seguridad frente a un usuario malicioso.
- Distintos procesos pueden tener su propio sistema de archivos raíz, que es un concepto parecido al que proporciona la herramienta `chroot` (explicado en la sección 2.1.1) [79].
- Los puntos de montaje se pueden hacer privados o compartidos.

[1] Los *magic links* son una clase especial de enlaces simbólicos que utiliza el *kernel* para representar información de ciertos *pseudo-filesystems* [84].

[2] *Mounting* es la acción de asociar un dispositivo de almacenamiento a una localización particular en el árbol de directorios [87]

[3] Es un tipo de *mount* que replica un árbol de directorios existente en un punto distinto. Una modificación en el *bind mount* provoca la misma modificación en el punto original, ya que ambos están apuntando a los mismos datos [88].

UTS NS

El siguiente *namespace*, el de UTS (*UNIX Time-Sharing*), que se incluyó en el año 2006, en la versión 2.6.19 del *kernel* de Linux. Este es el más sencillo, aunque no por ello innecesario, ya que aisla el *hostname* y el *domain-name* del contenedor [74, 13]. Estas características se pueden consultar mediante la llamada al sistema `uname` que devuelve dos identificadores: `nodename` y `domainname` [79].

Para cambiar estos nombres se puede utilizar las llamadas `sethostname` y `setdomainname` [79].

Su *flag* es `CLONE_NEWUTS`.

IPC NS

El *namespace* de IPC (*Inter Process Communication*) se implementó en el mismo año y versión de Linux que el anterior. Proporciona aislamiento en la compartición de memoria, en objetos System V IPC [86] y colas de mensajes POSIX [74, 79, 77].

Sin embargo, después de un `clone` no se aísla todo entre el proceso padre y el hijo: se siguen compartiendo señales, sockets, descriptores de archivos o sondeos de memoria (en inglés, *polling memory*) (aunque estos se pueden aislar con la ayuda de otros *namespaces*) [85].

Su *flag* correspondiente es `CLONE_NEWIPC`.

PID NS

El *namespace* de PID (*Process ID*) fue implementado sobre la versión de Linux 2.6.24 en el año 2008 [79].

Los procesos dentro de este *namespace* pertenecen a un árbol de procesos global que es visible únicamente por el *host*, y otro árbol propio de cada contenedor. De esta forma un mismo proceso puede tener varios PIDs dependiendo de en qué árbol se consulte (incluso puede tener el mismo PID que otro proceso en otro *namespace*)[4], como se puede comprobar en la figura 3.4 [74, 1].

El primer PID dentro del nuevo *namespace* siempre es el 1. Este proceso debería tener unas características únicas para poder funcionar como *init* y así poder tener más de un proceso en un contenedor [89, 170]:

- No maneja ninguna señal por defecto, por lo que son todas ignoradas al menos que se registre manualmente lo contrario (por eso los contenedores Docker, por ejemplo, no responden ante una señal de CTRL+C).
- Si otro proceso dentro del *namespace* muere antes que su hijo, el proceso `init` va a adoptar al hijo.

[4]Los *namespaces* se pueden anidar, permitiendo que exista un contenedor dentro de un contenedor, etc. Así, un mismo proceso puede tener un PID distinto en cada árbol. [74]

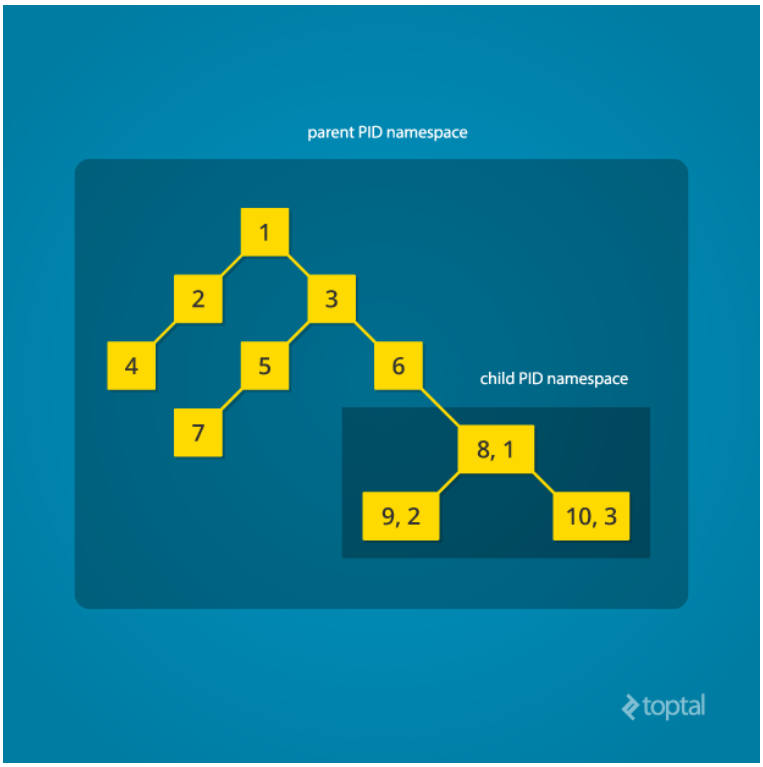

Figura 3.4: PIDs de un proceso perteneciente a distintos *PID namespaces*. Fuente: `https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces`

- Si muere, el *namespace* y todos los procesos que contiene se terminarán forzosamente.

Uno de los principales beneficios que presenta esta herramienta es que los contenedores se pueden migrar de una máquina a otra manteniendo el árbol de procesos [79].

Su *flag* es `CLONE_NEWPID`.

Network NS

El cuarto *namespace* que se añadió a Linux en el año 2009 en la versión 2.6.29 fue el *network namespace* [79]. Está considerado uno de los más complejos (también tuvo uno de los tiempos de desarrollo más grandes) [13].

Proporciona aislamiento de los recursos de red, donde cada contenedor puede tener su propio stack de recursos: tablas de enrutado, reglas de *iptables*, *sockets*, interfaces de red, puertos y más [76, 79].

En los contenedores se suele usar el modelo del «puente de red» (en inglés,

bridge)[5]. En este modelo, los contenedores tienen su propia interfaz virtual conectada a otra interfaz virtual del puente, que está en el *network namespace* global y proporciona tanto la conexión con el exterior como el enrutamiento apropiado a las interfaces virtuales que se conectan con los contenedores [91, 74, 77, 76, 79]. Esto se puede ver con más claridad en la imagen 3.5.

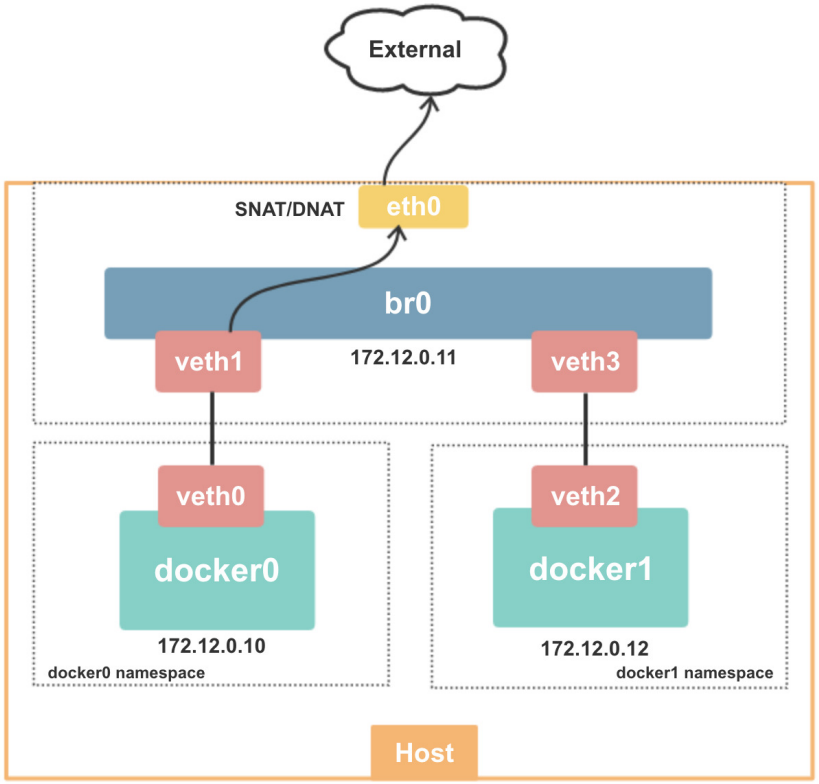

Figura 3.5: Ejemplo de un modelo con Docker utilizando un puente de red. Fuente: `https://dzone.com/articles/step-by-step-guide-establishing-container-networki`

La *flag* correspondiente es `CLONE_NEWNET`.

User NS

El *namespace* de usuarios se incluyó en Linux en la versión 3.8, en el año 2013, después de comenzarse a implementar en la versión 2.6.23 [79].

Antes de que existiera este *namespace* el root tenía UID 0 aunque las *capabilities*[6] le pusieran ciertas restricciones. Esta implementación permite que los

[5]De hecho, Docker utiliza este sistema, donde el puente "docker0" está siempre presente en todas las instalaciones de Docker [152].

[6]Las *capabilities* permiten dividir en muchas partes los privilegios propios de un usuario root. Estas se tratarán con mayor detenimiento en el apartado 3.1.4

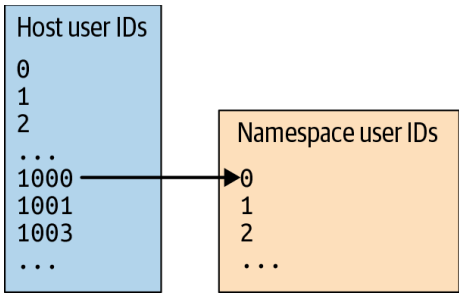

Figura 3.6: Mapeo de un usuario sin privilegios a un usuario root en el *namespace*. Fuente: `https://www.oreilly.com/library/view/container-security/9781492056690/ch04.html`

procesos puedan creer que están operando como root dentro de un contenedor con los privilegios que ello conlleva, pero fuera del contenedor tienen realmente los privilegios de un usuario común. Esta característica es crítica para la seguridad [13].

El UID dentro del *namespace* siempre se corresponderá con otro fuera de él, en el *host*. Este mapeo se puede hacer manualmente para elegir qué privilegios se obtienen dentro del contenedor.

Supongamos que mapeamos un proceso con UID 1000 en el *host* a un proceso con UID 0 en el *namespace*. En este caso, dentro del contenedor, podríamos realizar acciones con permisos de administrador, pero si quisiéramos acceder, por ejemplo, al archivo `/etc/shadow` (del *host*) no podríamos porque los permisos propios del *host* no nos lo permiten [93, 78].

En la figura 3.6 podemos ver gráficamente el mapeo de usuarios entre *namespaces*, en concreto el usuario UID 1000 del *host* correspondiéndose con el usuario UID 0 dentro de la tabla propia del *namespace*.

A diferencia del resto de *namespaces* vistos hasta ahora, para crear un nuevo *user namespace* no es necesario tener permisos de administrador [92].

La *flag* correspondiente es `CLONE_NEWUSER`.

Cgroup NS

Los *cgroups* o grupos de control son una funcionalidad del Linux Kernel que fue implementada en 2008 y que trataremos en este documento en la sección 3.1.2 (resumidamente, los *cgroups* implementan monitorización y limitación de los recursos que consume un grupo de procesos). El *namespace* relacionado con los grupos de control fue creado en 2016 en la versión Linux 4.6 [74].

Este *namespace* virtualiza la vista de los grupos de control que ven los procesos a través de la rutas `/proc/[pid]/cgroup` o `/proc/self/cgroup`. Sin esta restricción, un proceso podría observar los grupos de control globales a través de esas rutas [94, 1].

Esta virtualización tiene dos propósitos a mayores [94]:

- Facilita la migración de contenedores, ya que aísla unos grupos de control de otros ahorrando la necesidad de replicar los límites de recursos repartidos.
- Evita que un grupo de control acceda a los límites de otro superior a él:

  Imaginemos que no hemos implementado este *namespace*. Si creamos un grupo de control (que son representados mediante directorios) llamado `cg/1` que puede ser accedido por un usuario con UID 1000 y este mismo usuario crea un proceso con un nuevo grupo de control bajo el anterior, por ejemplo `cg/1/2`, este nuevo proceso podrá ver no sólo su directorio `2`, sino también el directorio del grupo superior, `1` (porque ambos los creó el mismo usuario). Con este *namespace* estaríamos aislando el *cgroup* del nivel inferior aunque el usuario perteneciente a ambos sea el mismo.

La *flag* correspondiente a este *namespace* es `CLONE_NEWCGROUP`.

Time NS

Por último, nos encontramos el *time namespace*, implementado en la versión de Linux 5.6, en el año 2020 [96].

Esta herramienta virtualiza dos relojes del sistema:

- **CLOCK_MONOTONIC**, un reloj «monotónico»[7] que no se puede ajustar y representa el tiempo que ha pasado desde un evento en el pasado [95, 97].
- **CLOCK_BOOTTIME**, que es como el anterior pero incluye los momentos en los que el equipo está suspendido [95].

Cuando un contenedor se migra de un nodo a otro, los relojes son restaurados de forma consistente partiendo siempre del tiempo que tenía el reloj antes de ser migrado [1].

Para finalizar con los *namespaces*, vamos a introducir las estructuras del *Linux Kernel* que permiten que éstos funcionen correctamente.

En primer lugar, es necesario saber que cada proceso está representado por una estructura llamada `task_struct`, que a su vez contiene otra estructura, `nsproxy`, que maneja los distintos *namespaces*[8] [1]:

[7] Que sólo se incrementa y no permite saltos.

[8] El *pid namespace* es una excepción, que se accede usando `task_active_pid_ns` [98]

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct group_namespace *cgroup_ns;
}
```

3.1.2. Control Groups

En este apartado veremos una característica del Linux Kernel que permite aplicar cuotas de una serie de recursos del *kernel* a un grupo de procesos [1, 99].

Paul Menage, el principal creador de los grupos de control, comentaba que, aunque existían formas para monitorizar y controlar procesos individuales, no había soporte para aplicar esas mismas operaciones a grupos de procesos, por lo que en 2006 implementó los *Process Containers* (que luego fueron renombrados a grupos de control o *cgroups* para distinguirlos de los contenedores actuales, que incluyen más funcionalidades de Linux). Estos son un *framework* basado en los mecanismos existentes del *kernel* para proporcionar una interfaz y un alcance más global a las operaciones de monitorización y control de procesos [29].

Por lo tanto, los grupos de control permiten repartir y asignar recursos entre un grupo de tareas o procesos de un sistema. Es posible monitorizar los grupos, denegar acceso a ciertos recursos e incluso reconfigurarlos de forma dinámica. Así, los administradores pueden obtener un gran control sobre los recursos del sistema incrementando la eficiencia global [100].

Los grupos de control ofrecen cuatro características principales [101, 99]:

- **Limitación de recursos**: permiten a un administrador asegurarse de que los programas que se están ejecutando en un sistema se mantengan dentro de unos límites en el uso de procesador, memoria, dispositivos de E/S, etc.
- **Priorización**: es una característica parecida a la anterior, pero se diferencia en que, independientemente de los recursos disponibles en un momento, siempre se va a dar preferencia a un proceso seleccionado.
- **Monitorización**: por lo general no suele estar activada esta característica en la mayoría de las versiones empresariales de Linux debido a que consume muchos recursos. Se puede activar para un particular grupo de procesos y de esta manera permite comprobar qué procesos están consumiendo cuántos recursos.

- **Control de procesos**: Existe una herramienta llamada *freezer* que permite paralizar o retomar grupos de procesos. Esto se puede utilizar para programar los recursos disponibles de la máquina. Es muy utilizado en el procesamiento por lotes [102].

Los *cgroups* se pueden activar montándolos en el sistema de archivos, de una forma similar a `/proc` o `/sys`. Muchas de las opciones que presenta esta herramienta se pueden configurar en el directorio `/sys/fs/cgroup`, aunque existen ciertas librerías que facilitan estas tareas como `libcgroup` [13].

CGroups v2

Para comprender correctamente esta herramienta es necesario conocer ciertas definiciones:

- **Cgroup**: grupo de tareas al que se le asocia uno o más subsistemas [103].
- **Subsistema o *resource controller***: representa un único recurso, como la memoria o el tiempo de CPU [100].
- **Jerarquía**: es el conjunto de todos los *cgroups* plasmados en un árbol. Cada proceso del sistema está en uno de los *cgroups*. Cada nodo (o sea, cada cgroup) del árbol tiene asociado uno o más subsistemas. En la versión 2 de los grupos de control sólo existe una única jerarquía [104].

En la figura 3.7 se muestra un ejemplo de jerarquía. En este ejemplo podemos ver dos cgroups principales (Group_1 y Group_2) con dos subsistemas asociados a cada uno, el segundo tiene dos grupos hijos a mayores. Cada grupo se añade creando un nuevo directorio dentro de la carpeta raíz `/sys/fs/cgroup`. Por defecto no hay ningún subsistema asociado a un grupo, pero se pueden añadir fácilmente escribiendo en los archivos de la ruta `/sys/fs/cgroup/`. En el ejemplo de la figura, se añadirían los subsistemas mediante las siguientes instrucciones [105, 106]:

```
echo "+memory +io" > /sys/fs/cgroup/Group_1/cgroup.subtree_control
echo "+memory +pids" > /sys/fs/cgroup/Group_2/cgroup.subtree_control
echo "+io" > /sys/fs/cgroup/Group_2/Group_2.1/cgroup.subtree_control
echo "-pids" > /sys/fs/cgroup/Group_2/Group_2.2/cgroup.subtree_control
```

Cabe destacar que los grupos heredan los subsistemas y límites establecidos por los padres, de tal forma que el Group_2.1 del ejemplo contaría con tres subsistemas asociados (*memory*, *pids* e *io*) mientras que el Group_2.2 tendría únicamente uno (*memory*) ya que ha disociado el controlador *pids* creado por el padre [104].

Antes de terminar con los grupos de control, vamos a introducir los distintos tipos de subsistemas o controladores que existen en la versión 2 de los *cgroups* [106].

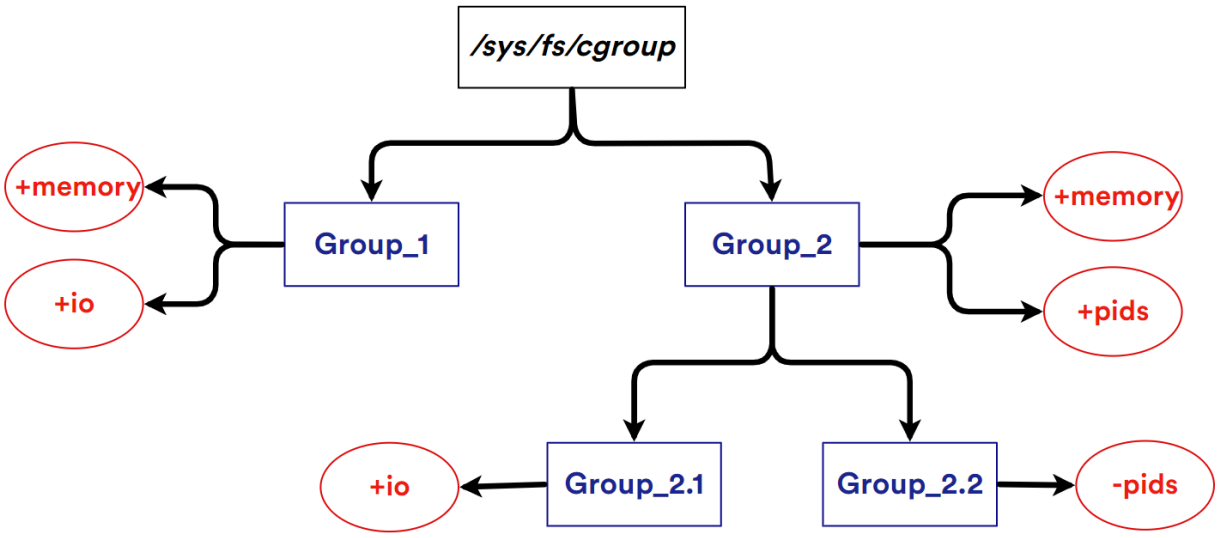

Figura 3.7: Ejemplo de una jerarquía de grupos de control

- **CPU**: regula la distribución de los ciclos del procesador. Implementa prioridad y límites de ancho de banda para las *scheduling policies*.
- **Memory**: regula la distribución de la memoria. Implementa tanto límites como protección (un mínimo de memoria) para evitar el mal funcionamiento de los procesos.
- **IO**: regula la distribución de recursos de entrada y salida. Implementa prioridades y ancho de banda absoluto.
- **PID**: permite a un grupo de control impedir que cualquier proceso suyo pueda realizar una llamada `fork` o `clone` si se cumple el límite establecido. Esto puede evitar algunos ataques tipo *fork bomb* antes de que haya problemas en el uso de la memoria.
- **Cpuset**[9]: proporciona un mecanismo para restringir la CPU y nodos de memoria que se le asignan a unos procesos de un grupo de control.
- **Device Controller**: gestiona los dispositivos de archivos. Incluye tanto la creación de nuevos dispositivos de archivos, usando `mknod`, como el acceso a dichos dispositivos.

[9]Los sistemas más modestos pueden operar con eficiencia dejando al sistema compartir los recursos de memoria y CPU, sin embargo, en sistemas más grandes como en aplicaciones HPC se benefician de este tipo de restricciones asignando *subsets* del sistema de manera más cuidadosa [108, 107]

- **RDMA (Remote Direct Memory Access)**[10]: regula la distribución y contabilidad de los recursos RDMA.
- **Huge TLB**: permite limitar el uso de la HugeTLB [112] en cada grupo.

3.1.3. Union Filesystem

Los sistemas de archivos por capas permiten compartir archivos en el disco, lo que supone un ahorro de espacio. Esta compartición se produce en la memoria [1].

Los **Union Filesystems** son un tipo de sistema de archivos por capas, que implementan un *union mount*[11]. Permite a los archivos y directorios de sistemas de archivos distintos, llamados ramas, estar superpuestos, formando un único sistema de archivos [1, 114].

OverlayFS

Docker, actualmente, utiliza un ejemplo de UnionFS llamado Overlay2. Esta tecnología utiliza tres capas [1, 115]:

1. **Base** (Read Only): es la capa donde van los archivos base. En términos de Docker u otras tecnologías de contenedores parecidas, se correspondería con la imagen.
2. **Overlay** (Vista de usuario): es la capa donde opera el usuario. En un principio ofrece una vista de la capa *base* y permite operar sobre ella, tanto añadiendo como modificando archivos. Cuando se escriben los cambios no se guardan en esta capa, sino en la siguiente, en *diff*.
3. **Diff**: es la capa donde se guardan los cambios realizados en la capa anterior.

   Una vez hechos los cambios, la capa *overlay* ofrecerá una vista como la unión de la capa *base* y *diff* (con los archivos de *diff* remplazando a los de *base* en el caso de que haya conflictos).

En el caso de Docker, se crean varias capas *base* y una capa *overlay*. Las diferentes capas *base* pueden ser compartidas entre contenedores del mismo *host* [1].

Una de las características principales de los UnionFS es que usan la técnica llamada COW (*Copy On Write*), que, en este ámbito, permite reducir el consumo de copias sin modificar: si un fichero existe en la capa *base* y otra capa quiere leerlo, simplemente lee el original. Cuando una capa quiera modificar el archivo es cuando debe de copiarla y modificarla en la propia capa que la necesite [116].

[10] Es una tecnología que permite a dos ordenadores de una red intercambiar información en la memoria principal sin necesidad de incluir a la CPU, caché o sistema operativo de ambos equipos. Parte de DMA (Direct Memory Access [109, 111]) [110]

[11] Es una manera de combinar varios directorios en uno de tal forma que los contenidos aparecen combinados [113].

3.1.4. Capabilities

El usuario con UID 0 es el root y tiene el control completo del sistema. Esto puede traer problemas recurrentes de seguridad (con las Jails ya intentaron confinar el root omnipotente).

La solución propuesta en las *capabilities* es dividir los permisos que tiene dicho usuario en varias particiones, cada una de ellas es una *capability* [13]. Esta herramienta se implementó en la versión 2.2 del *kernel* de Linux [117].

Un buen ejemplo del uso de una *capability* podría ser la posibilidad de permitir a un binario poder crear *raw sockets* como con la instrucción `ping` mediante la *capability* `CAP_NET_RAW` sin necesidad de asignarle el resto de privilegios que ofrece el usuario root [13].

Las *capabilities* están mantenidas en conjuntos, que se representan como *bit masks*. Existen dos tipos de *capabilities*.

- Asociadas a **procesos**: los procesos tienen unas *capabilities* agrupados en cinco conjuntos donde la información no se guarda por proceso sino por hilo [13, 117, 119, 120, 118]:
  - ***Permitted***: es el conjunto de *capabilities* que el proceso puede asumir.
  - ***Inheritable***: es el conjunto de *capabilities* que el proceso puede heredar de su proceso padre.
  - ***Effective***: es el conjunto de *capabilities* que el proceso tiene asociadas inicialmente y le pide al *kernel* permiso para usar.
  - ***Bounding***: es el conjunto de *capabilities* que se pueden añadir a los conjuntos anteriores. Así, si el proceso tiene la *capability* `CAP_SETCAP` (que da el poder de asignar *capabilities*) pero la nueva *capability* que se quiere asignar no está en este conjunto, no se podrá añadir a los otros últimos tres conjuntos explicados.
  - ***Ambient***: es el conjunto de *capabilities* que son añadidas al conjunto *permitted* y *effective* cuando se llama a `execve` (es decir, cuando se quiere ejecutar un programa y se remplazan los contenidos del actual proceso por el nuevo).
- Asociadas a **binarios**: Estos conjuntos, junto con los conjuntos propios de los procesos (de hilos realmente) determinan las *capabilities* finales con las que se ejecuta el programa [117]:
  - ***Permitted***: este conjunto de *capabilities* se añaden automáticamente al conjunto *permitted* del hilo.
  - ***Inheritable***: las *capabilities* de este conjunto se añaden al conjunto *Inheritable* del hilo.

CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_FSETID
CAP_FOWNE
CAP_MKNOD
CAP_NET_RAW
CAP_SETGID
CAP_SETUID
CAP_SETFCAP
CAP_SETPCA
CAP_NET_BIND_SERVICE
CAP_SYS_CHROOT
CAP_KILL
CAP_AUDIT_WRITE

Tabla 3.1: Lista de *capabilities* que presenta un contenedor Docker por defecto

- ***Effective***: no es realmente un conjunto, sino un bit. Si este bit está puesto, durante un `execve` todas las *capabilities* que tiene el hilo en el conjunto *permitted* se convierten en efectivos en este conjunto.

En la práctica, los contenedores no necesitan todos los privilegios que ofrece el root. Así que, realmente, los usuarios root dentro de los contenedores tienen asignadas algunas *capabilities* para ofrecer únicamente ciertos permisos y restringir otros potencialmente inseguros [75].

En la tabla 3.1 se muestra la lista de *capabilities* que presentan por defecto los contenedores Docker.

3.1.5. Pivot_root

En el capítulo 2 indicamos la importancia que tuvo `chroot` ya que permitía aparentar que cambiaba el directorio raíz de un proceso y de sus hijos [122]. Es una herramienta que se puede utilizar para crear contenedores y ofrecer una vista limitada del sistema de archivos, sin embargo, no es segura, así que es mejor utilizar la alternativa `pivot_root`.

La función completa es la siguiente:

```
int pivot_root(const char *new_root, const char *put_old);
```

El problema principal de `chroot` es que se aplica únicamente a un sólo proceso, así que técnicamente se sigue pudiendo acceder a los directorios superiores de la nueva raíz. Una forma muy sencilla es simplemente mediante la instrucción `chroot /proc/1/root` [123].

Sin embargo, `pivot_root` mueve el punto de montaje raíz al directorio *put_old* y convierte a *new_root* en el nuevo punto de montaje raíz. Al hacer esto, está cambiando tanto el directorio raíz del proceso llamador como el de todos sus hilos e hijos [73].

Esto, soluciona las brechas de seguridad de `chroot` porque se aplica al *mount namespace*, así que está cambiando en el *namespace* del contenedor la posibilidad de acceder a la antigua raíz al moverla a `put_old` (sin mover la raíz del host ya que está en otro *namespace*) [125, 124].

3.2. Introducción a los «Container Runtimes»

Un término muy usado en el ámbito de los contenedores es *container runtime*. Sin embargo, el significado de este término puede variar dependiendo del proyecto o comunidad donde se consulte.

Un *container runtime* es la herramienta o capa responsable de que el contenedor se ejecute correctamente (sin incluir los procesos que están dentro del contenedor) [127]. Estas tareas que permiten cumplir el objetivo de ejecutar un contenedor se pueden separar en varios niveles, por eso, en este documento vamos a separar los *container runtime* en *low-level* y *high-level*.

Algunos ejemplos de *low-level* pueden ser **runc, crun, gvisor o kata-runtime**, mientras que entre los ejemplos de *high-level* podemos encontrarnos con **containerd o CRI-O**.

Cabe destacar que en muchas referencias se trata a Docker como un *container runtime* (refiriéndose a Docker con su significado de *daemon* más su CLI). Llegados a este punto, podemos entender perfectamente que se pueda considerar de esta forma, ya que su función principal es gestionar los contenedores y las imágenes (definición de *high-level container runtime*). Sin embargo, en este documento se va a colocar en un escalón superior, en los *container engines*.

Esta decisión se ha tomado en base a dos razones: la primera es que una de las características principales que hacen a esta herramienta más popular que containerd o CRI-O entre los usuarios es su capacidad de facilitar la comunicación con el usuario. La segunda razón es que delega gran parte del trabajo en otro *high-level runtime*, que por defecto es containerd. Dicho esto reiteramos que se podría considerar perfectamente como un *runtime*. Esto mismo opina Insu Jang, ingeniero de software de TmaxSoft Inc: "Docker Engine (dockerd) puede ser un (*high-level*) *container runtime* dada la existencia de sus drivers; driver de almacenamiento, driver de red, etc. Honestamente, no estoy seguro de que Docker engine por sí mismo, sin containerd, pueda ser considerado un *container runtime*" [153].

De forma similar, CRI-O o containerd se podrían llegar a considerar *container engines* si se usan a través de plugins CLI que permitan al usuario interactuar debidamente con la herramienta.

En resumen, la forma de referiste a estas tecnologías no es universal y, como

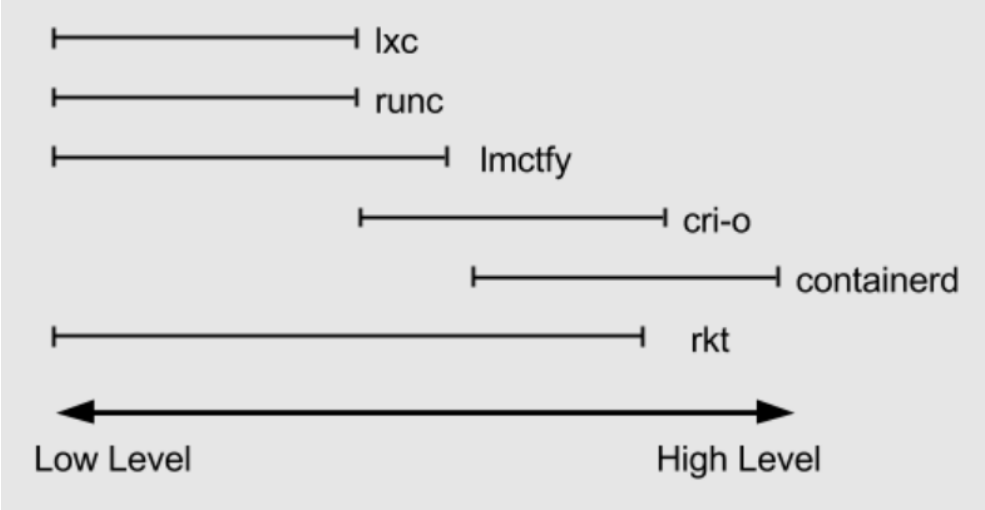

Figura 3.8: Nivel al que opera cada *container runtime*. Fuente: `https://insujang.github.io/2019-10-31/container-runtime`

hemos dicho, dependiendo de la comunidad y el proyecto, las etiquetas pueden cambiar.

Finalmente, una forma de entender gráficamente el porqué de que estos términos sean tan ambiguos es observando la figura 3.8, que muestra, de forma muy general y subjetiva, el nivel al que operan algunos ejemplos de *container runtimes*.

3.2.1. Low-level Container Runtime

En este documento nos referiremos a los *low-level container runtime* para hablar de las capas que proporcionan unas utilidades básicas como crear *namespaces* y comenzar el proceso de encapsular una aplicación en un contenedor [126], es decir, a las herramientas que se comunican directamente con el *kernel*.

runc (parte del stack de Docker)

Runc es básicamente un CLI que se encarga de crear contenedores y ejecutarlos según la especificación que se le proporcione. Esta especificación sigue el estándar OCI [128].

Runc no entiende el concepto de imágenes: esta herramienta espera que se le especifique un *OCI bundle*[12] y un archivo de configuración llamado `config.json` [129].

Los estándares OCI ya se introdujeron en este mismo documento en la sección 2.1.11, sin embargo, en aquel capítulo no había suficiente contexto. En dicho apartado se había comentado la existencia de dos estándares: de **imagen** y de

[12]Un *bundle* es un conjunto de archivos que contiene todos los datos necesarios para que un *runtime* pueda realizar todas las operaciones necesarias para crear y ejecutar un contenedor [130]

runtime. En este caso, runc sólo implementa la segunda especificación ya que no utiliza imágenes, sino *bundles*.

En los anteriores párrafos comentábamos que runc es básicamente un CLI. ¿Cómo puede ser un *runtime* si realmente es un simple CLI? Para entender esto debemos de remontarnos a los inicios de Docker, cuando utilizaba LXC (2.1.7) para crear los contenedores. En la versión 0.9 esto dejó de ser así y creó el famoso proyecto OCI. Por aquel entonces Google estaba creando su propia versión de *runtime* llamado LMCTFY (2.1.9). Este último proyecto se canceló dada la popularidad de Docker y ambos comenzaron a crear una librería propia que sustituyera a LXC: *libcontainer* [131, 132].

Hoy en día, el proyecto runc incluye esta librería como pieza básica y fundamental, por eso se consideran una única capa. De hecho, *libcontainer* se puede encontrar en un directorio del repositorio de runc en Github [134].

crun

Otro ejemplo de *low-level container runtime* que sigue el estándar OCI es el caso de **crun**.

Fue creado por Giuseppe Scrivano que opinaba que no era lógico que la mayoría de herramientas sobre el ecosistema de los contenedores estuvieran escritas en Go. Éste es un lenguaje multihilo por defecto, lo que no es recomendable para utilizar en un modelo de `fork/exec`. Su solución se basa en crear un *runtime* en C, que es un lenguaje más apropiado para manejar herramientas de bajo nivel [136, 135].

El resultado es un *runtime* muy ligero que utiliza los *cgroups v2* (no todas las herramientas en el ámbito de los contenedores han realizado la transición a la versión 2). Además, el propio creador muestra unas pruebas de rendimiento en su página de Github [137] donde compara mediente Podman tanto runc como crun y, a priori, parece que los resultados son convincentes a favor de crun.

Aún así, el estándar que utiliza Docker sigue siendo runc.

Otros Sandboxed Runtimes

Las anteriores tecnologías comparten sistema operativo con el *host*. Esto implica gran eficiencia pero mayor vulnerabilidad, especialmente en el entorno de la nube donde distintos clientes comparten las mismas máquinas físicas. A partir de estos hechos surgieron nuevos proyectos por parte de IBM, Google, Amazon y OpenStack que pretendían crear *sandboxed containers*, una solución que ofrece mayor aislamiento sobre los contenedores [139, 138]:

- **Google gVisor**: es utilizado por GCP (*Google Computer Platform*). Esta herramienta encapsula las aplicaciones interceptando las llamadas al sistema y desviándolas a su propia implementación del *kernel* en espacio de

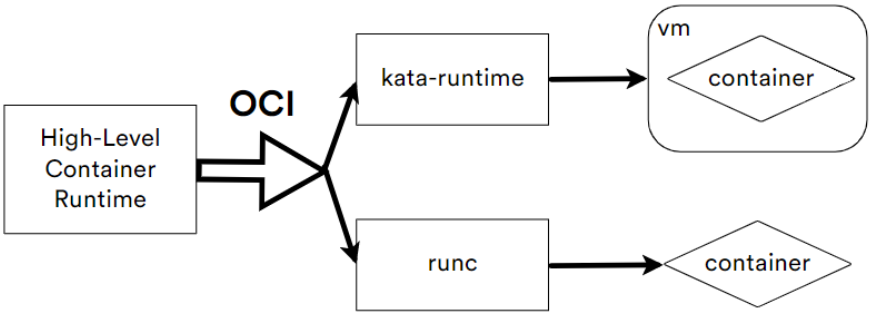

Figura 3.9: Comparación genérica de *sandboxed runtime* y runc utilizando como ejemplo kata-containers.

usuario. Runsc es la pieza fundamental de gVisor e implementa el estándar OCI; al fin y al cabo, esta pieza es realmente el *low-level runtime*, y gVisor envuelve a runsc.

- **Amazon Firecracker**: es un VMM (1.3.3) que crea una máquina virtual muy ligera, llamada MicroVM. Esta MV es útil específicamente para entornos *multi-tenant*. La VMM depende de KVM y cada instancia se ejecuta en espacio de usuario.
- **OpenStack Kata**: el caso de Kata es especial, no sólo implementa ambas especificaciones OCI sino que también implementa CRI (*Container Runtime Interface*), que veremos en la próxima sección. Esto quiere decir que no sólo se puede considerar *low-level*, sino también *high-level container runtime*.

En la figura 3.9 se puede ver a grandes rasgos la diferencia entre un *sandboxed runtime* como Kata-runtime y uno normal como runc.

Finalmente en la figura 3.10 se muestra una pequeña comparación de estos tres *sandboxed runtimes runtimes*.

3.2.2. High-level Container Runtime

Los *high-level runtimes* son los responsables de la gestión y transporte de las imágenes de los contenedores, así como de desempaquetarlas y enviarlas a los *low-level runtimes*, abstrayéndolos de esas actividades [140, 141]. Dos de estos *high-level container runtime* son containerd y CRI-O.

containerd (parte del stack de Docker)

Al igual que runc, forma parte del stack de Docker. Es un estándar en la industria y es famoso por su portabilidad y facilidad de uso [142]. Implementa

Security Sandbox Solution	Implementation Way	Isolation Way	Performance, Lightweight	Universality、Flexibility	Applicable scenarios	Vendor Who Is Using
Google gVisor	User Mode Kernel	Syscall Interception	Unsatisfactory IO、networking performance	Only support limited syscalls、Not universal	Serverless	Google
AWS Firecracker	Lightweight Virtualization	Virtualization	High	General, Incomplete virtualization, Not universal enough	Serverless	aws、ucloud
Kata Containers	Lightweight Virtualization	Virtualization	High	Universal, Supporting a variety of lightweight hypervisors	Multiple demand scenarios	Baidu、Huawei、Ant Financial、Alibaba

Figura 3.10: Comparación entre gVisor, Kata y Firecracker. Fuente: `https://miro.medium.com/max/624/1*Ov829Xc4mWNu1Kw-M9o7iA.png`

ambas especificaciones OCI: es capaz de desempaquetar imágenes OCI y llamar a un *low-level runtime* siguiendo la especificación OCI [129].

Las tareas principales que puede realizar esta herramienta son [129]:

- Hacer *pull* y *push* a las imágenes.
- Gestionar el almacenamiento de los contenedores.
- Gestionar las redes y sus interfaces.
- Llamar a un *low-level container runtime* con los parámetros necesarios para que ejecute el contenedor.

Containerd es un *daemon*, aunque dispone de varios plugins que aumentan sus funcionalidades en gran medida como una interfaz de línea de comandos.

Una parte fundamental de containerd es el **shim**. Este elemento también aparece en Kubernetes en la capa superior a Docker *daemon* (ver figura 3.2), pero no realizan la misma tarea, aunque ambos son *shims*[13].

Containerd-shim es una implementación que permite separar a los *low-level runtimes* de containerd. Cuando éste último quiere llamar a runc (o cualquier otro *low-level*) crea una nueva instancia de runc por cada contenedor que se ordena crear. De esta forma, cuando runc logra ejecutar el contenedor se termina a sí mismo: a partir de ahora el containerd-shim se convierte en el nuevo padre del contenedor y pasa a tomar las responsabilidades que ello implica. Esto permite que no haya un programa pesado como los *runtimes* en forma de *daemon* pendiente de un contenedor [144, 145, 129] y que, si Docker falla, se mantenga el STDIO abierto y el contenedor ejecutándose [147]. Esto está ilustrado en la figura 3.11.

[13]En el ámbito de los ordenadores, es un componente de software que actúa como puente entre distinas APIs o como una capa que aporta compatibilidad compatibilidad (como en este caso, que facilita la comunicación con otros programas) [143]

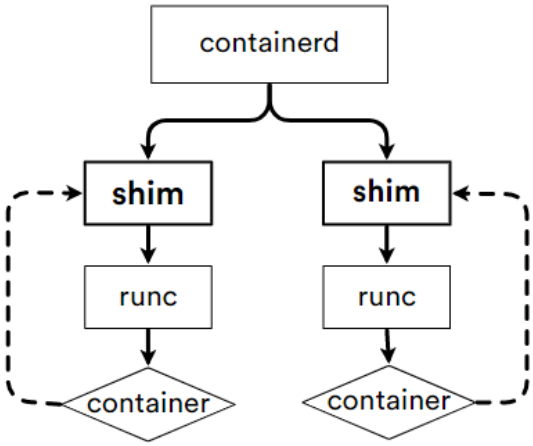

Figura 3.11: Interacción de containerd con containerd-shim y runc.

Finalmente, indicar que containerd implementa CRI mediante un plugin oficial, lo que le da una gran versatilidad al permitir que un orquestador como Kubernetes pueda comunicarse con él sin pasar por Docker [146, 143].

CRI-O y la interfaz CRI

Antes de hablar sobre CRI-O es fundamental entender lo que significa CRI (*Container Runtime Interface*). **CRI es la API que utiliza Kubernetes para hablar con los *high-level runtimes***. El proyecto partió del hecho de que Kubernetes es un orquestador (que trataremos con más detenimiento en la sección 3.5) que delega parte del trabajo a los *runtimes*. De esta forma, al establecer una API que describe cómo interactúa esta tecnología con todos ellos aumenta en gran medida su interoperabilidad, permitiendo establecer un «estándar» (al menos dentro del stack de Kubernetes) y que cada *runtime* pueda gestionar los contenedores a su manera, siempre y cuando respeten este CRI [143].

CRI está basado en gRCP, un tipo de RPC (Remote Procedure Call) que permite intercomunicación entre lenguajes gracias al uso de *protocol buffers* [149].

En el apartado anterior habíamos comentado que containerd disponía de una serie de plugins. Uno de ellos permite la comunicación con Kubernetes mediante CRI, lo que hace que pueda seguir usándose con el orquestador.

Por otro lado, CRI-O es una herramienta (que actúa como *daemon* [55]) creada por Red Hat para soportar de forma nativa la comunicación mediante CRI, sirviendo de puente entre Kubernetes y un *low-level runtime* compatible con el estándar OCI. Al igual que containerd, puede comunicarse cumpliendo el estándar OCI con, por ejemplo, runc y realizar las típicas tareas de los demás *high-level runtimes* [148, 131, 143, 141].

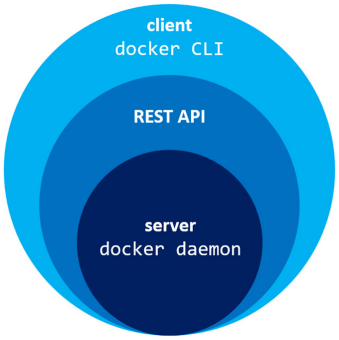

Figura 3.12: Arquitectura del Docker *daemon*. Fuente: `https://nickjanetakis.com/blog`

En cuanto al rendimiento de CRI-O y containerd, el estudio encabezado por Lennart Espe llamado "Performance Evaluation of Container Runtimes" [149] analiza distintas combinaciones entre containerd y CRI-O utilizando tanto runc como runsc (de gVisor). Los resultados muestran que, en cuanto a rendimiento general, CRI-O más runc resulta ser la mejor opción.

Cabe destacar que CRI-O está muy unido a Kubernetes, delegando la gestión de las imágenes a otro *runtime* o *engine* como Podman (que también es de Red Hat) [153].

3.3. Container Engine

Una vez presentadas las capas inferiores (ambos *container runtimes* y las primitivas de Linux) es mucho más sencillo contextualizar los *container engines*.

El *container engine* por excelencia es el propio de Docker, formado por el Docker *daemon* más sus APIs y CLI. Por lo tanto, será el ejemplo a tratar en esta sección.

Estos componentes que lo forman, y que se muestran en la imagen 3.12, son:

- **Docker *daemon***: es el servicio que se ejecuta en la parte del *host* o servidor y escucha peticiones API. Esta capa gestiona objetos de Docker como imágenes o volúmenes. Es la primera herramienta que comienza la creación del contenedor. Además, dado que los *runtimes* están aislados del *container engine*, este último se puede reiniciar o actualizar sin la necesidad parar los contenedores [154, 155, 157, 156].

- **API REST**: el *daemon* escucha las peticiones de los clientes a través de esta API REST, que utiliza *UNIX sockets* [154, 155].

- **Interfaz en línea de comandos (CLI)**: es la forma con la que los usuarios, actuando como clientes, se comunican con el *daemon* a través de la

API comentada [155].

3.4. Otros componentes: imágenes y registros

En los últimos apartados hemos hablado sobre los elementos y capas principales que componen los contenedores y, más en concreto, el stack de Docker.

Sin embargo, existen otros componentes y «objetos» que no forman parte de ninguna de las capas anteriores pero siguen siendo elementos fundamentales del ecosistema, como las imágenes y registros.

A lo largo del documento se ha ido hablando repetidamente de imágenes. Este concepto es muy parecido al mismo término que se usa en el ecosistema de las máquinas virtuales, de ahí que seguramente no hubiera demasiados problemas para entender las explicaciones que se iban dando en el trabajo.

Una imagen es un archivo binario inmutable que contiene el código, librerías, dependencias e información que necesita un *runtime* para crear un contenedor.

El hecho de que sean archivos inmutables, es decir, de sólo lectura, permite que equipos de desarrollo puedan probar software en un entorno estable y uniforme, ya que representan un entorno en un momento y condiciones específicas que puede ser representado las veces que sea necesario mediante una misma imagen.

Además, en el caso de Docker, existe un archivo que describe cómo se va a construir una imagen específica, este archivo se llama `Dockerfile`. Este fichero de texto contiene una lista de instrucciones que se van a ejecutar a la hora de crear la imagen, es decir, cuando se ejecute el comando `docker build`. Normalmente, el Dockerfile utiliza como base otra imagen anterior y se le añaden nuevas instrucciones para configurar la nueva de la forma deseada. Así, una imagen se crea normalmente mediante el apilamiento de varias imágenes base [158, 159].

Por otro lado, un **registro** es básicamente un sistema o repositorio que se encarga de **almacenar y distribuir imágenes**.

Existen dos tipos de registros [160]:

- **Públicos**: son adecuados para pequeños equipos o usuarios individuales que buscan obtener una imagen para una función específica sin importarles demasiado la privacidad y seguridad. Docker Hub es el ejemplo más popular y cuenta con más de 100.000 imágenes que cualquier usuario puede descargar [161].

- **Privados**: cuentan con una mayor privacidad y seguridad ya que sólo se comparten con los usuarios deseados, normalmente entre un equipo de trabajo.

Para acabar, la figura 3.13 muestra cómo actúan algunos de los componentes principales de Docker, incluyendo los registros e imágenes.

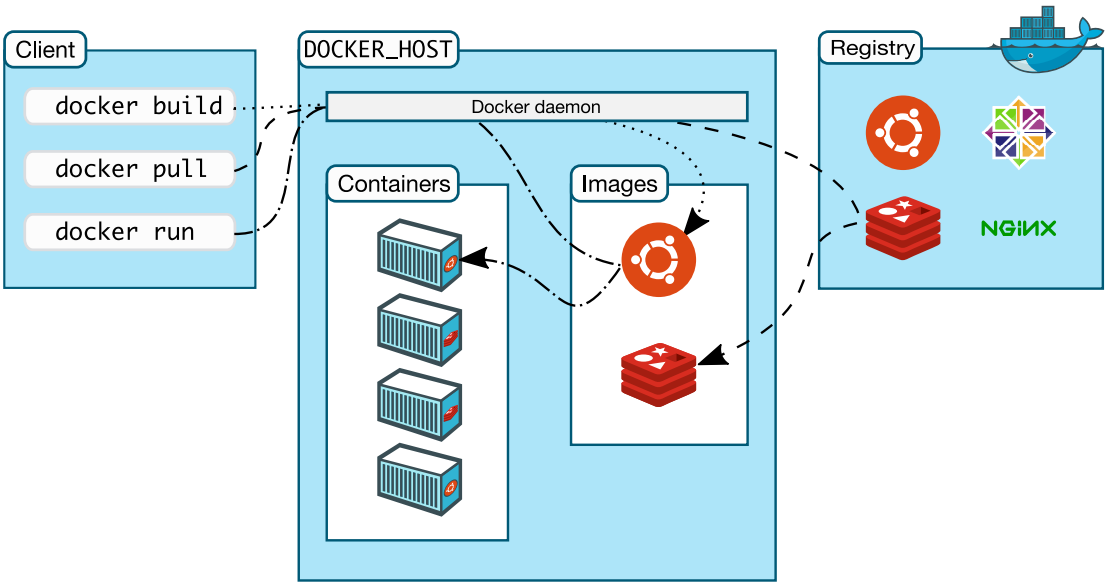

Figura 3.13: Arquitectura del Docker *daemon*. Fuente: `https://docs.docker.com/get-started/overview/`

3.5. Orquestadores

Administrar un gran número de contenedores es una tarea muy complicada, por eso surgieron ciertas tecnologías que permiten automatizar el proceso de crear, desplegar y escalar contenedores [162].

Estas herramientas son los orquestadores (en inglés, *orchestrators*). Su objetivo principal es facilitar la gestión de contenedores localizados en muchos *hosts*. Todo esto facilita a los equipos de DevOps integrar sus flujos CI/CD [163].

En concreto, las tareas que suelen realizar los orquestadores son [162, 163, 164]:

- Configurar y programar contenedores.
- Proporcionar el aprovisionamiento y despliegue adecuados de los contenedores.
- Asegurar la disponibilidad de los servicios.
- Escalar los contenedores para conseguir un balance sobre los flujos de trabajo de la infraestructura.
- Controlar los recursos repartidos entre contenedores.
- Balancear la carga y el tráfico, así como ejercer tareas de enrutado.
- Monitorizar la salud de los contenedores.
- Proporcionar seguridad en las interacciones entre los contenedores.

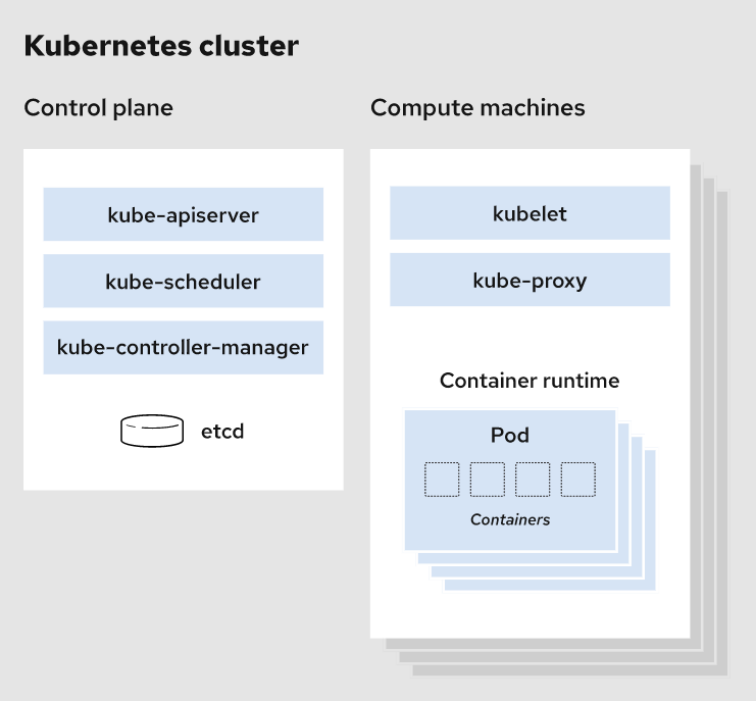

Figura 3.14: Componentes de un *cluster* de Kubernetes. Fuente: `https://www.redhat.com/en/topics/containers/kubernetes-architecture`.

- Exponer los servicios al exterior.

3.5.1. Kubernetes

Kubernetes es el estándar de facto en la actualidad en lo que a orquestadores se refiere. Es una herramienta portable y de código libre desarrollada originalmente por Google.

Sus tres principios son: **seguridad**, **facilidad de uso** y **extensibilidad** (que se pueda utilizar con varios proveedores).

Cuando se despliega una instancia de Kubernetes se está creando realmente un *cluster*, que se compone de dos elementos fundamentales, el **control plane** y las **compute machines** o **nodos**, tal y como se puede observar en la imagen 3.14.

El *control plane* es el núcleo de Kubernetes y es el encargado de que todo el servicio funcione correctamente asegurando que se esté ejecutando un número adecuado de contenedores con los recursos suficientes. Está formado por cuatro componentes [165, 166]:

- API *server*: maneja las peticiones internas y externas. Se puede establecer comunicación con este componente a través de una API REST (es el método que usa la CLI `kubectl`).

- *Scheduler*: controla la salud de los *pods* y los recursos que necesita. Es el encargado de añadir un nuevo contenedor al *pod* si fuera necesario.
- *Controller Manager*: controla que el *cluster* funcione correctamente, en realidad este gestor está formado por varios controladores, cada uno con una función específica.
- *etcd*: es una base de datos «clave-valor» distribuida. Permite almacenar la configuración y el estado de los *clusters* de Kubernetes.

Por otro lado están los nodos (por lo general existen varios en un mismo *cluster*) que se pueden escalar en el caso de que sea necesario. Estos nodos están compuestos por [165, 166]:

- *Pods*: representa una instancia de una aplicación, se corresponde con un sólo contenedor o con varios altamente acoplados.
- *Container runtime/engine*: se pueden utilizar varios como Docker, containerd o CRI-O.
- *Kubelet*: es la aplicación que controla a los contenedores de un mismo *pod*. Se encarga de comunicarse con el *control plane*.
- *Proxy*: facilita los servicios de red.

Ahora que ya hemos explicado tanto Docker como Kubernetes, podemos volver a la imagen 3.2 y aclarar un par de detalles.

La primera es que el Docker *daemon* no se comunica con Kubernetes, sino que realmente se está comunicando con cada *kubelet* de un *cluster* de Kubernetes.

La segunda es que ahora podemos entender bien qué es **Docker-shim**. Este elemento es una solución propuesta por la comunidad de Kubernetes para que Kubernetes soportara a Docker como *container runtime* ya que Docker no implementa CRI de por sí. Sin embargo, en el año 2020 la comunidad de Kubernetes anunció que no va a seguir manteniendo este plugin, así que el uso de Kubernetes más Docker tiene un futuro incierto [167, 168].

Kubernetes y sus *container runtimes*

En la figura 3.15 podemos ver una comparación entre los componentes o capas normalmente utilizados en el stack de Kubernetes dependiendo del *runtime* o *engine* que se utilice.

Está claro que CRI-O no implementa tantos saltos entre componentes como las otras dos opciones lo que tiene sus obvias ventajas. Sin embargo, la mejor forma de evaluar cuál de estas opciones es más óptima es mediante un *benchmark*. En el 2017, Kunal Kushwaha publicó en NTT OSS Center [169] los resultados de un

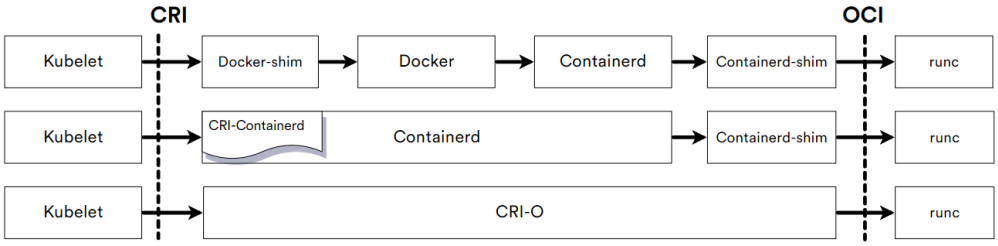

Figura 3.15: Componentes o capas usadas según el *container runtime* que se use con Kubernetes.

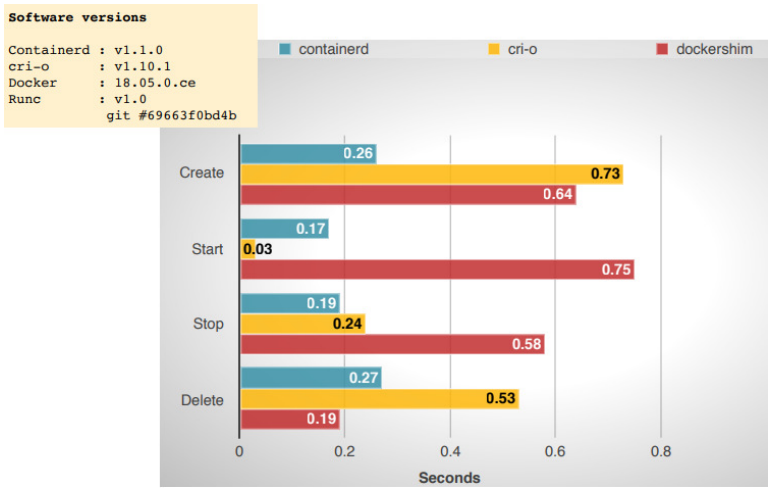

Figura 3.16: Comparación de rendimiento (en segundos) entre Docker, containerd y CRI-O usando Kubernetes como orquestador y runc como *low-level container runtime*. Fuente `https://events19.linuxfoundation.org/wp-content/uploads/2017/11/How-Container-Runtime-Matters-in-Kubernetes_-OSS-Kunal-Kushwaha.pdf`.

análisis en el que comparaba los tres *runtimes* mencionados en la imágen 3.15 desde Kubernetes utilizando como *low-level container engine* runc.

Las pruebas consistían en crear, comenzar, parar y borrar, con cuatro hilos, cincuenta contenedores. Los resultados se muestran en la figura 3.16 que le asigna un tiempo a cada una de las tareas mencionadas. En ella se puede comprobar que containerd es el que mejor resultado tiene en general.

Capítulo 4

Conclusión y posibles ampliaciones

La informática se encuentra en un momento en el que están surgiendo constantemente nuevas tecnologías, mientras que otras tantas se van quedando obsoletas. En el ecosistema de los contenedores esto se exagera todavía más, ya que el mundo de la filosofía DevOps y de la nube lleva creciendo los últimos años a pasos agigantados. Esto nos lleva a que la información que podemos encontrar sobre cada tecnología esté cambiando y ampliándose constantemente: mismamente, en los primeros cuatro meses del año 2021 se han creado una gran cantidad de blogs y artículos en páginas web sobre los temas que hemos tratado en este documento con información muy útil.

Otro punto a tener en cuenta es el ya comentado en el capítulo 3.2 donde se explicaba que dependiendo de la comunidad o proyecto en el que se busque la información pueden existir diferentes formas de referirse a un mismo elemento.

4.1. Cumplimiento de objetivos

Al comienzo del proyecto se propusieron una serie de objetivos que debía cumplir una vez estuviera finalizado. A continuación, se muestra cómo se han cumplido dichos objetivos de forma organizada:

- Se han introducido los principales conceptos relacionados con la virtualización, añadiendo, también, la diferencia entre las máquinas virtuales y los contenedores.

- Se ha explicado la historia y evolución de los contenedores añadiendo algunas alternativas que existen en la actualidad como Docker, Podman o Singularity.

- Se ha propuesto una forma de dividir las capas y componentes que forman

el stack de las tecnologías de contenedores tomando como referencia el stack de Docker.

  - Se ha explicado cómo funciona cada capa comenzando desde la de nivel inferior, es decir, desde las funcionalidades que proporciona el *kernel* de Linux.
  - Se han ido explicando las distintas capas a medida que se subía de nivel en el stack poniendo como ejemplo de cada una sus alternativas más populares, incluyendo las que forman el stack de Docker.

- Además, se han introducido los orquestadores utilizando como referencia Kubernetes.

- Por último, se han presentado los anteriores puntos de una forma más atractiva y resumida en una página web [172] incluyendo los dos tutoriales propuestos: un tutorial más explicativo en lenguaje Go y otro en Bash (aunque este último es muy corto ya que Bash presenta ciertas instrucciones que facilitan enormemente la tarea).

4.2. Recomendaciones de ampliación

En cuanto a el capítulo 2.1, las tecnologías más importantes que allanaron el camino a las tecnologías de contenedores que se usan actualmente son principalmente las Jails y Zones. Los documentos donde se describen mejor estas tecnologías son precisamente los que las presentan; en el caso de las Jails, "Jails: Confining the omnipotent root" [18] y en el de las Zones, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads" [27]. Sería interesante ampliar las descripciones de ambas tecnologías dado su interés histórico. Además, existió otra tecnología llamada Rkt bastante innovadora en su momento y, aunque en la actualidad ya está obsoleta, podría ser una posible ampliación al ser uno de los precursores de Docker.

Por otro lado, sería muy interesante añadir la descripción de otros componentes del kernel de Linux que utilizan los contenedores. En este documento se explican los principales, dejando muy de lado la seguridad y los componentes implicados como *apparmor* o *seccomp*.

Finalmente, hay que destacar que la memoria no se ha enfocado ni orquestadores y otras herramientas por encima de los *container engines* ni en la parte práctica de cómo gestionarlos, por lo que podría ser buena decisión ampliar la información del documento añadiendo para qué se usan otros orquestadores como Docker Swarm, Nomad o Apache Mesos, además de otras tecnologías como Docker Compose.

Apéndice A

Licencia

Este documento está bajo la licencia MIT. Se trata de una licencia permisiva que, mientras se incluya la copia original de la licencia, se puede realizar cualquier cambio o modificación en el documento.

Esto permite que se pueda ampliar, usar o modificar fácilmente la información siempre y cuando se atribuya esta fuente. Lo que es realmente útil en un documento como este en el que se recopile y presente información con fines educativos.

Bibliografía

[1] Shashank Mohan Jain. *Linux Containers and Virtualization. A Kernel Perspective*, Apress. New York, 2020.

[2] Virtual Machines. Artículo de IBM Cloud Education (`https://www.ibm.com/cloud/learn/virtual-machines`). Consultado el 4 de abril del 2021.

[3] Hypervisors. Artículo de IBM Cloud Education (`https://www.ibm.com/cloud/learn/hypervisors`). Consultado el 4 de abril del 2021.

[4] Pekka Enberg. The Evolution and Future of Hypervisors. Artículo de Medium (`https://medium.com/@penberg/the-evolution-and-future-of-hypervisors-999f568f9a5d`). Consultado el 4 de abril del 2021.

[5] Containers vs. VM: What's the difference? Artículo de IBM Cloud e IBM Cloud Team (`https://www.ibm.com/cloud/blog/containers-vs-vms`). Consultado el 4 de abril del 2021.

[6] What are containers? Artículo de Netapp (`https://www.netapp.com/devops-solutions/what-are-containers/`). Consultado el 4 de abril del 2021.

[7] Richard Harris & Joel Nelson. Containers 101: What is a container technology, what is Kubernetes and why do you need them? Artículo de Rackspace (`https://www.rackspace.com/blog/containers-101`). Consultado el 4 de abril del 2021.

[8] Containers. Artículo de IBM Cloud Education (`https://www.ibm.com/cloud/learn/containers`). Consultado el 4 de abril del 2021.

[9] Containerization. Artículo de IBM Cloud Education (`https://www.ibm.com/cloud/learn/containerization`). Consultado el 4 de abril del 2021.

[10] Contenedores en Google. Artículo de Google (`https://cloud.google.com/containers`). Consultado el 5 de abril del 2021.

[11] Disk footprint. (2019, 4 de abril). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 23:21, junio 17, 2021 desde https://en.wikipedia.org/w/index.php?title=Disk_footprint&oldid=890923887

[12] Larry Seltzer. What containers and cloud native are and why theyre hot. Artículo de Medium (https://medium.com/enterprise-nxt/what-containers-and-cloud-native-are-and-why-theyre-hot-691cb9f015d4). Consultado el 6 de abril del 2021.

[13] Aaron Grattafiori. Understanding and Hardening Linux Containers (versión 1.1, Junio de 2016). Paper de nccgroup (https://www.nccgroup.com/ae/our-research/understanding-and-hardening-linux-containers/). Consultado el 6 de abril del 2021.

[14] Virtualization. Artículo de IBM Cloud Education (https://www.ibm.com/cloud/learn/virtualization-a-complete-guide) Consultado el 6 de abril del 2021.

[15] Peter Rombouts. Container Cons: The complexities associated with Containers. Artículo de Sogeti Labs (https://labs.sogeti.com/container-cons-the-complexities-associated-with-containers/). Consultado el 6 de abril del 2021.

[16] Stephen Guyton. Containers & Containerization - Pros and Cons. Artículo de Atomic Object (https://spin.atomicobject.com/2019/05/24/containerization-pros-cons/). Consultado el 6 de abril del 2021.

[17] Randal, A. (2020). The ideal versus the real. Revisiting the history of virtual machines and containers. *ACM Computing Surveys (CSUR) 53*(1), 1-31. Disponible en (https://arxiv.org/abs/1904.12226). Consultado el 6 de abril del 2021.

[18] Kamp, P. H., & Watson, R. N. (2000, May). Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (Vol. 43, p. 116). Disponible en la web de SANE (http://www.sane.nl/events/sane2000/papers/). Consultado el 6 de abril del 2021.

[19] Rostislav M. Georgiev. The Story of Containers. Artículo de VMware (https://blogs.vmware.com/opensource/2018/02/27/the-story-of-containers/). Consultado el 6 de abril del 2021.

[20] Imesh Gunaratne. The Evolution of Linux Containers and Their Future. Artículo de DZone (https://dzone.com/articles/evolution-of-linux-containers-future). Consultado el 6 de abril del 2021.

[21] Ozan Eren. Something Missed?: History of Container Technology. Artículo de Medium (`https://oziie.medium.com/something-missed-history-of-container-technology-e978f202464a`). Consultado el 6 de abril del 2021.

[22] Containers: From the origins to Docker. Artículo de CriticalCase (`https://www.criticalcase.com/blog/containers-from-the-origins-to-docker.html`). Consultado el 6 de abril del 2021.

[23] Ell Marquez. The History of Container Technology. Artículo de ACloudGuru (`https://acloudguru.com/blog/engineering/history-of-container-technology`). Consultado el 6 de abril del 2021.

[24] Tim Hildred. The History of Containers. Artículo de RedHat (`https://www.redhat.com/en/blog/history-containers`). Consultado el 6 de abril del 2021.

[25] Rani Osnat. A Brief History of Containers From the 1970s Till Now. Artículo de Aqua Blog (`https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016`). Consultado el 6 de abril del 2021.

[26] PapersWeLove. Bryan Cantrill on Jails and Solaris Zones. Vídeo de YouTube (`https://www.youtube.com/watch?v=hgN8pCMLI2U`). Consultado el 7 de abril del 2021.

[27] Price, D., & Tucker, A. (2004, November). Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA* (Vol. 4, pp. 241-254). Disponible en la web de USENIX (`https://www.usenix.org/legacy/publications/library/proceedings/lisa04/tech/full_papers/price/price.pdf`). Consultado el 18 de junio del 2021.

[28] Linux-VServer. (2020, 16 de abril). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 00:05, junio 18, 2021 desde `https://en.wikipedia.org/w/index.php?title=Linux-VServer&oldid=951395068`

[29] Menage, P. B. (2007, June). Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium* (Vol. 2, pp. 45-57). Disponible en (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.798&rep=rep1&type=pdf#page=45`). Consultado el 18 de junio del 2021.

[30] Michael Kerrisk. Stepping closer to practical containers: "syslog" namespaces. Artículo de LWN (`https://lwn.net/Articles/527342/`). Consultado el 8 de abril del 2021.

[31] namespaces(7). Artículo del manual de Linux (https://man7.org/linux/man-pages/man7/namespaces.7.html). Consultado el 8 de abril del 2021.

[32] The History of Virtualization and Containerization. Artículo de K8S Tutorials (https://kubernetestutorials.com/kubernetes-beginners-tutorials/the-history-of-virtualization-and-containerization/). Consultado el 10 de abril del 2021.

[33] Michael Maximilien. The history of IBM's contributions to Cloud Foundry, part 1. Artículo de IBM Developer Blog (https://developer.ibm.com/technologies/paas/blogs/history-cloud-foundry-1/). Consultado el 10 de abril del 2021.

[34] Maksim Zhylinski. Cloud Foundry Containers: The Difference Between Warden, Docker and Garden. Artículo de ProgrammerSought (https://www.programmersought.com/article/73433959225/). Consultado el 10 de abril del 2021.

[35] Differences Between DEA and Diego Architectures. Artículo de CloudFoundry Docs (https://docs.cloudfoundry.org/concepts/diego/dea-vs-diego.html). Consultado el 10 de abril del 2021.

[36] Docker vs CoreOS Rkt. Artículo de UpGuard (https://www.upguard.com/blog/docker-vs-coreos). Consultado el 14 de abril del 2021.

[37] Docker Features. Artículo de JavaTpoint (https://www.javatpoint.com/docker-features). Consultado el 14 de abril del 2021.

[38] Priya Pedamkar. Rkt vs Docker. Artículo de EDUCBA (https://www.educba.com/rkt-vs-docker/). Consultado el 15 de abril del 2021.

[39] Christopher Tozzi. Why is Docker so Popular? Explaining the Rise of Containers and Docker. Artículo de Channel Futures (https://www.channelfutures.com/open-source/why-is-docker-so-popular-explaining-the-rise-of-containers-and-docker). Consultado el 15 de abril del 2021.

[40] Docker Hub Quickstart. Artículo de Docker Docs (https://docs.docker.com/docker-hub). Consultado el 15 de abril del 2021.

[41] Open Container Initiative. (2021, 6 de febrero). Wikipedia, *La enciclopedia libre* Fecha de consulta: 00:27, junio 18, 2021 desde (https://en.wikipedia.org/w/index.php?title=Open_Container_Initiative&oldid=1021384679)

[42] About the Open Container Initiative. Artículo de Open Container Initiative (`https://opencontainers.org/about/overview`). Consultado el 15 de abril del 2021.

[43] Linux Foundation (`https://www.linuxfoundation.org`). Consultado el 15 de abril del 2021.

[44] Arthur Busser. From Docker to OCI: What is a container? Artículo de Padok (`https://www.padok.fr/en/blog/container-docker-oci`). Consultado el 15 de abril del 2021.

[45] Bin Chen. Open Container Initiative (OCI) Specifications. Artículo de Medium (`https://alibaba-cloud.medium.com/open-container-initiative-oci-specifications-375b96658f55`). Consultado el 15 de abril del 2021.

[46] Systemd. (2021, 9 de abril). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 00:52, junio 18, 2021 desde (`https://es.wikipedia.org/w/index.php?title=Systemd&oldid=134645600`).

[47] rkt vs other projects. Artículo de Rocket Docs (`https://rocket.readthedocs.io/en/latest/Documentation/rkt-vs-other-projects/`). Consultado el 17 de abril del 2021.

[48] appc. Repositorio de Appc Container en Github (`https://github.com/appc/spec`). Consultado el 17 de abril del 2021.

[49] Adriaan de Jonge. Moving from Docker to rkt. Artículo de Medium (`https://medium.com/@adriaandejonge/moving-from-docker-to-rkt-310dc9aec938`). Consultado el 17 de abril del 2021.

[50] Docker frequently asked questions [FAQ]. Artículo de Docker Docs (`https://docs.docker.com/engine/faq`). Consultado el 17 de abril del 2021.

[51] rkt. Repositorio de Github de Rkt [obsoleto] (`https://github.com/rkt/rkt`). Consultado el 17 de abril del 2021.

[52] What is rkt? Artículo de OpenShift (`https://www.openshift.com/learn/topics/rkt`). Consultado el 17 de abril del 2021.

[53] Podman. Repositorio de Github de Containers (`https://github.com/containers/podman`). Consultado el 17 de abril del 2021.

[54] Podman. Página oficial de Podman (`https://podman.io/`). Consultado el 17 de abril del 2021.

[55] Ivan Velichko. A journey from containerization to orchestration and beyond. Artículo del blog iximiuz (https://iximiuz.com/en/posts/journey-from-containerization-to-orchestration-and-beyond/). Consultado el 19 de abril del 2021.

[56] DevopsCurry. Looking for an alternative to docker in 2021? Podman could be your solution!. Artículo de Medium (https://medium.com/devopscurry/looking-for-an-alternative-to-docker-in-2021-podman-could-be-your-solution-d99425c6b9ac). Consultado el 19 de abril del 2021.

[57] Dan Walsh. Crictl vs Podman. Artículo de OpenShift (https://www.openshift.com/blog/crictl-vs-podman). Consultado 19 de abril del 2021.

[58] San Diego Computer Center. Introduction to Singularity: Containers for Scientific and High-Performance Computing. Vídeo de YouTube (https://www.youtube.com/watch?v=vEjLuX0ClN0). Consultado el 19 de abril del 2021.

[59] What is High-Performance Computing? Artículo de NetApp (https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/). Consultado el 19 de abril del 2021.

[60] Why is Singularity used as opposed to Docker in HPC and what problems does it solve? Post de Reddit (https://www.reddit.com/r/docker/comments/7y2yp2/why_is_singularity_used_as_opposed_to_docker_in/). Consultado el 19 de abril del 2021.

[61] Gerber, L. (2018). Containerization for HPC in the Cloud: Docker vs Singularity-A Comparative Performance Benchmark. Disponible en (http://www.diva-portal.org/smash/get/diva2:1277794/FULLTEXT01.pdf). Consultado el 19 de abril del 2021.

[62] Using the CRI-O Container Engine. Artículo de OpenShift (https://docs.openshift.com/container-platform/3.11/crio/crio_runtime.html). Consultado el 20 de abril del 2021.

[63] podman vs CRI-O vs RunC. Post de RedHat Learning Community. Hilo de RedHat Learning Community (https://learn.redhat.com/t5/Containers-DevOps-OpenShift/podman-vs-CRI-O-vs-RunC/td-p/9639). Consultado el 20 de abril del 2021.

[64] Guruprasad Padmanabha Chikklore. LXC vs LXD vs Docker - Evolution of the Container Ecosystem. Artículo de Devon (https://www.devonblog.com/continuous-delivery/lxc-vs-lxd-vs-docker-evolution-of-the-container-ecosystem/). Consultado el 20 de abril del 2021.

[65] Docker vs LXC. Artículo de UpGuard (`https://www.upguard.com/blog/docker-vs-lxc`). Consultado el 20 de abril del 2021.

[66] Docker vs LXC....What is the consensus on this? Hilo de Reddit (`https://www.reddit.com/r/linuxadmin/comments/7s9ctv/docker_vs_lxcwhat_is_the_consensus_on_this/`). Consultado el 20 de abril del 2021.

[67] Hemant Jain. LXC and LXD: Explaining Linux Containers. Artículo de sumo logic (`https://www.sumologic.com/blog/lxc-lxd-linux-containers/`). Consultado el 20 de abril del 2021.

[68] Página web de Linux Containers (`https://linuxcontainers.org/`). Consultado el 20 de abril del 2021.

[69] Christopher Tozzi. Comparing OpenVZ and LXD Linux System Container Platforms. Artículo de Contaner Journal (`https://containerjournal.com/features/comparing-openvz-lxd-linux-system-container-platforms/`). Consultado el 21 de abril del 2021.

[70] Craig Wilhite, Heidi Lohr, Elizabeth Ross, Alexey Brodkin, Taylor Brown, Sarah Cooley and Maira Wenzel. Windows Container Essentials: Isolation modes. Artículo de Microsoft Docs (`https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container`). Consultado el 21 de abril del 2021.

[71] rkt. Repositorio de Github de Rkt [obsoleto] (`https://github.com/rkt/rkt/`). Consultado el 26 de abril del 2021.

[72] capabilities(7). Artículo del manual de Linux (`https://man7.org/linux/man-pages/man7/capabilities.7.html`). Consultado el 27 de abril del 2021.

[73] pivot_root(2). Artículo del manual de Linux (`https://man7.org/linux/man-pages/man2/pivot_root.2.html`). Consultado el 27 de abril del 2021.

[74] Desmitifying Containers. Repositorio de Github de saschagrunert (`https://github.com/saschagrunert/demystifying-containers`). Consultado el 29 de abril del 2021.

[75] Docker security. Artículo de Docker Docs (`https://docs.docker.com/engine/security/`). Consultado el 29 de abril del 2021.

[76] Docker. Cgroups, namespaces, and beyond: what are containers made from? Vídeo de YouTube (`https://www.youtube.com/watch?v=sK5i-N34im8`). Consultado el 29 de abril del 2021.

[77] Mahmud Ridwan. Separation Anxiety: A tutorial for Isolating Your System with Linux Namespaces. Artículo de Toptal (https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces). Consultado el 29 de abril del 2021.

[78] Ifeany Ubah. A deep dive into Linux namespaces. Serie de artículos de la web ifeani (http://ifeanyi.co). Consultado el 29 de abril del 2021.

[79] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. Artículo de LWN (https://lwn.net/Articles/531114/). Consultado el 29 de abril del 2021.

[80] clone(2). Artículo del manual de Linux (https://man7.org/linux/man-pages/man2/clone3.2.html). Consultado el 29 de abril del 2021.

[81] unshare(1). Artículo del manual de Linux (https://man7.org/linux/man-pages/man1/unshare.1.html). Consultado el 29 de abril del 2021.

[82] setns(2). Artículo del manual de Linux (https://man7.org/linux/man-pages/man1/unshare.1.html). Consultado el 29 de abril del 2021.

[83] ioctl(2). Artículo del manual de Linux (https://man7.org/linux/man-pages/man2/ioctl_ns.2.html). Consultado el 29 de abril del 2021.

[84] symlink(7). Artículo del manual de Linux (https://man.archlinux.org/man/symlink.7.en). Consultado el 30 de abril del 2021.

[85] Jean-Tiare Le Bigot. Introduction to Linux namespaces (https://blog.yadutaf.fr/). Consultado el 30 de abril del 2021.

[86] sysvipc(7). Artículo del manual de Linux (https://man7.org/linux/man-pages/man7/svipc.7.html). Consultado el 30 de abril del 2021.

[87] What is meant by mounting a device in Linux? Hilo de Stackexchange (https://unix.stackexchange.com/questions/3192/what-is-meant-by-mounting-a-device-in-linux). Consultado el 1 de mayo del 2021.

[88] What is a bind mount? Hilo de Stackexchange (https://unix.stackexchange.com/questions/198590/what-is-a-bind-mount). Consultado el 1 de mayo del 2021.

[89] Vish (Ishaya) Abrams. The curious case of Pid Namespaces. Artículo de Hackernoon (https://hackernoon.com/the-curious-case-of-pid-namespaces-1ce86b6bc900). Consultado el 1 de mayo de 2021.

[90] Bridge Networking Deep Dive. Repositorio de Github de Peng Xiao (`https://github.com/xiaopeng163/docker-k8s-lab/blob/master/docs/source/docker/bridged-network.rst`). Consultado el 1 de mayo del 2021.

[91] Freynman Zhou. Step-by-Step Guide: Establishing Container Networking. Artículo de dzone (`https://dzone.com/articles/step-by-step-guide-establishing-container-networki`). Consultado el 1 de mayo del 2021.

[92] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. Artículo de LWN (`https://lwn.net/Articles/532593`). Consultado el 1 de mayo del 2021.

[93] Philipp Schmied. LINUX CONTAINER PRIMITIVES: USER NAMESPACES. Artículo de Schutzwerk (`https://www.schutzwerk.com/en/43/posts/namespaces_04_user/`). Consultado el 1 de mayo del 2021.

[94] cgroup_namespaces(7). Artículo del manual de Linux (`https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html`). Consultado el 2 de mayo del 2021.

[95] time_namespace. Artículo del manual de Linux (`https://man7.org/linux/man-pages/man7/time_namespaces.7.html`). Consultado el 2 de mayo del 2021.

[96] Michael Larabel. It's Finally Time: The Time Namespaces Support Has Been Added To The Linux 5.6 Kernel. Artículo de Phoronix (`https://www.phoronix.com/scan.php?page=news_item&px=Time-Namespace-In-Linux-5.6`). Consultado el 2 de mayo del 2021.

[97] Difference between CLOCK REALTIME and CLOCK MONOTONIC? Hilo de Stackoverflow (`https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic`). Consultado el 2 de mayo del 2021.

[98] nsproxy.h. Código fuente de kernel de Linux (`https://elixir.bootlin.com/linux/latest/source/include/linux/nsproxy.h`). Consultado el 2 de mayo del 2021.

[99] Cgroups. (2021, 10 de junio). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 15:36, junio 18, 2021 desde (`https://en.wikipedia.org/w/index.php?title=Cgroups&oldid=1027879091`).

[100] Milan Navrátil, Peter Ondrejka, Eva Majoršinová, Martin Prpič, Rüdiger Landmann y Douglas Silas. RESOURCE MANAGEMENT GUIDE. Artículo de Red Hat Customer Portal (`https://access.redhat.`

com/documentation/en-us/red_hat_enterprise_linux/6/html-single/resource_management_guide/index). Consultado el 3 de mayo del 2021.

[101] Steve Ovens. A Linux sysadmin's introduction to cgroups. Artículo de Red Hat (https://www.redhat.com/sysadmin/cgroups-part-one). Consultado el 3 de mayo del 2021.

[102] Cgroup Freezer. Artículo de la documentación del kernel de Linux (https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/freezer-subsystem.html). Consultado el 3 de mayo del 2021.

[103] Menage, P., Jackson, P., & Lameter, C. (2008). Cgroups. Disponible en (https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt). Consultado el 3 de mayo del 2021..

[104] Marc Richter. World domination with cgroups in RHEL 8: welcome cgroups v2!. Artículo de Red Hat Blog (https://www.redhat.com/en/blog/world-domination-cgroups-rhel-8-welcome-cgroups-v2). Consultado el 3 de mayo del 2021.

[105] Chris Down. cgroupv2: Linux's new unified control group hierarchy (QCON London 2017). Vídeo de YouTube (https://www.youtube.com/watch?v=ikZ8_mRotT4). Consultado el 3 de mayo del 2021.

[106] Tejun Heo. Control Group v2. Artículo de la documentación del kernel de Linux (https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html). Consultado el 3 de mayo del 2021.

[107] cpuset(7). Artículo del manual de Linux (https://man7.org/linux/man-pages/man7/cpuset.7.html). Consultado el 3 de mayo del 2021.

[108] Derr S., Jackson P., Lameter C., Menage P. & Seto H. CPUSETS. Artículo de la documentación del kernel de Linux (https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cpusets.html). Consultado el 3 de mayo del 2021.

[109] Direct Memory Access. (2021, 12 de junio). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 15:49, junio 18, 2021 desde (https://en.wikipedia.org/w/index.php?title=Direct_memory_access&oldid=1028262250).

[110] Remote Direct Memory Access. (2021, 30 de enero). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 15:52, junio 18, 2021 desde (https://en.wikipedia.org/w/index.php?title=Remote_direct_memory_access&oldid=1003823417).

[111] James Alan Miller. Remote Direct Memory Access (RDMA). Artículo de Search Storage, TechTarget (`https://searchstorage.techtarget.com/definition/Remote-Direct-Memory-Access`). Consultado el 3 de mayo del 2021.

[112] R. Krishnakumar. HugeTLB - Large Page Support in the Linux Kernel. Artículo de Linux Gazette (`https://linuxgazette.net/155/krishnakumar.html`). Consultado el 3 de mayo del 2021.

[113] Union mount. (2021, 24 de mayo). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 15:55, junio 18, 2021 desde (`https://en.wikipedia.org/w/index.php?title=Union_mount&oldid=1024874785`).

[114] UnionFS. (2021, 10 de enero). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 15:55, junio 18, 2021 desde (`https://en.wikipedia.org/w/index.php?title=UnionFS&oldid=999568012`).

[115] Eli Uriegas. How Docker images work: Union File Systems for dummies (`https://www.terriblecode.com/blog/how-docker-images-work-union-file-systems-for-dummies/`). Consultado el 5 de mayo del 2021.

[116] About storage drivers. Artículo de Docker Docs (`https://docs.docker.com/storage/storagedriver/`). Consultado el 5 de mayo del 2021.

[117] capabilities(7). Artículo del manual de Linux (`https://man7.org/linux/man-pages/man7/capabilities.7.html`). Consultado el 5 de mayo del 2021.

[118] Linux Capabilities. Artículo de Hacktricks (`https://book.hacktricks.xyz/linux-unix/privilege-escalation/linux-capabilities`). Consultado el 5 de mayo del 2021.

[119] Philipp Schmied. Linux Container Basics: capabilities. Artículo de Schutzwek (`https://www.schutzwerk.com/en/43/posts/linux_container_capabilities/`). Consultado el 5 de mayo del 2021.

[120] Adrian Mouat. Linux Capabilities: Why They Exist and How They Work. Artículo de Container Solutions (`https://blog.container-solutions.com/linux-capabilities-why-they-exist-and-how-they-work`). Consultado el 5 de mayo del 2021.

[121] Henryk Plötz. Understanding Capabilities in Linux. Artículo del blog ploetzli (`https://blog.ploetzli.ch/2014/understanding-linux-capabilities/`). Consultado el 5 de abril del 2021.

[122] Chroot. (2021, 17 de enero). Wikipedia, *La enciclopedia libre*. Fecha de consulta: 17:08, junio 18, 2021 desde (https://en.wikipedia.org/w/index.php?title=Chroot&oldid=1029039532).

[123] get out of a jail free. Hilo cPanel (https://forums.cpanel.net/threads/jailshell-get-out-of-jail-free.171490/). Consultado el 5 de mayo del 2021.

[124] Linux containers in a few line sof code. Hilo de Hacker News (https://news.ycombinator.com/item?id=23165157). Consultado el 5 de mayo del 2021.

[125] How does chroot-escape protection in LXC implemented. Hilo de Stackoverflow (https://stackoverflow.com/questions/23882087/how-does-chroot-escape-protection-in-lxc-implemented). Consultado el 5 de mayo del 2021.

[126] Ivan Velichko. A journey from containerization to orchestration and beyond. Artículo del blog iximiuz (https://iximiuz.com/en/posts/journey-from-containerization-to-orchestration-and-beyond/). Consultado el 8 de mayo del 2021.

[127] Ian Lewis. Container Runtimes Part 1: An Introduction to Container Runtimes. Artículo del blog ianlewis (https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r). Consultado el 8 de mayo del 2021.

[128] runc. Repositorio de GitHub de Open Containers Initiative (https://github.com/opencontainers/runc). Consultado el 9 de mayo del 2021.

[129] Alexander Holbreich. Docker components explained. Artículo del blog alexander.holbreick (https://alexander.holbreich.org/docker-components-explained/). Consultado el 9 de mayo del 2021.

[130] Filesystem Bundle. Repositorio de Github de Open Containers Initiative (https://github.com/opencontainers/runtime-spec/blob/master/bundle.md). Consultado el 9 de mayo del 2021.

[131] Evan Baker. A Comprehensive Container Runtime Comparison. Artículo de Capitalone (https://www.capitalone.com/tech/cloud/container-runtime/). Consultado el 9 de mayo del 2021.

[132] Juan Antonio. LibContainer Overview. Github Pages de jancorg (http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/). Consultado el 9 de mayo del 2021.

[133] Antoine Beaupré. Desmitifying container runtimes. Artículo de LWN (`https://lwn.net/Articles/741897/`). Consultado el 9 de mayo del 2021.

[134] libcontainer. Repositorio de Github de Open Container Initiative (`https://github.com/opencontainers/runc/tree/master/libcontainer`). Consultado el 9 de mayo del 2021.

[135] Walsh D., Rothberg V. & Scrivano G. An introduction to crun, a fast and low-memory footprint container runtime. Artículo de Red Hat (`https://www.redhat.com/sysadmin/introduction-crun`). Consultado el 9 de mayo del 2021.

[136] crun. Repositorio de Github de Containers (`https://github.com/containers/crun`). Consultado el 9 de mayo del 2021.

[137] Giuseppe Scrivano. running Podman as root. Repositorio gist de Github (`https://gist.github.com/giuseppe/aa7dd1f14d3eb235a993e5334e653b36`). Consultado el 9 de mayo del 2021.

[138] Jay Chen. Making Containers More Isolated: An Overview of Sandboxed Container Technologies. Artículo de Paloalto Network (`https://unit42.paloaltonetworks.com/making-containers-more-isolated-an-overview-of-sandboxed-container-technologies/`). Consultado el 10 de mayo del 2021.

[139] Li Ning. Exploration and Practice of Performance Tuning for Kata Containers 2.0. Artículo de Medium (`https://medium.com/kata-containers/exploration-and-practice-of-performance-tuning-for-kata-containers-2-0-85055d29e8b5`). Consultado el 10 de mayo del 2021.

[140] Ian Lewis. Container Runtimes Part 3 High-Level Runtimes. Artículo del blog ianlewis (`https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes`). Consultado el 10 de mayo del 2021.

[141] Karthikeyan Govindaraj. Understanding Kubernetes:2. Artículo de Medium (`https://medium.com/@github.gkarthiks/understanding-kubernetes-2-efe439efb9ce`). Consultado el 10 de mayo del 2021.

[142] containerd. Repositorio de Github de containerd (`https://github.com/containerd/containerd`). Consultado el 10 de mayo del 2021.

[143] Tom Donohue. The differences between Docker, containerd, CRI-O and runc. Artículo de Tutorial Works (`https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/`). Consultado el 11 de mayo del 2021.

[144] Mohit Goyal. Going Down the Rabbit Hole of Docker Engine...- shim. Artículo de mohitgoyal.co (https://mohitgoyal.co/2021/04/06/going-down-the-rabbit-hole-of-docker-engine-shim/). Consultado el 11 de mayo del 2021.

[145] Avijit Sarkar. Docker and OCI Runtimes. Artículo de Medium (https://medium.com/@avijitsarkar123/docker-and-oci-runtimes-a9c23a5646d6). Consultado el 11 de mayo del 2021.

[146] Anish Nath. container runtime runc,containerd,rkt,docker,cri. Artículo de 8gwifi (https://8gwifi.org/docs/containers.jsp). Consultado el 11 de mayo del 2021.

[147] Tim West. Diving Deeper Into Runtimes: Kubernetes, CRI, and Shims. Artículo de threat stack (https://www.threatstack.com/blog/diving-deeper-into-runtimes-kubernetes-cri-and-shims). Consultado el 11 de mayo del 2021.

[148] Ian Lewis. Container Runtimes Part 4: Kubernetes Container Runtimes & CRI. Artículo del blog ianlewis (https://www.ianlewis.org/en/container-runtimes-part-4-kubernetes-container-run). Consultado el 12 de mayo del 2021.

[149] Espe, L., Jindal, A., Podolskiy, V., & Gerndt, M. (2020). Performance Evaluation of Container Runtimes. In CLOSER (pp. 273-281). Disponible en (https://www.researchgate.net/profile/Anshul-Jindal-2/publication/341483813_Performance_Evaluation_of_Container_Runtimes/links/5fd0085792851c00f85f1de9/Performance-Evaluation-of-Container-Runtimes.pdf). Consultado el 12 de mayo del 2021.

[150] Fran S. ¿Qué es un VPS? Todo lo que necesitas saber sobre servidores virtuales. Artículo de Hostinger (https://www.hostinger.es/tutoriales/que-es-un-vps). Consultado el 12 de mayo del 2021.

[151] Introduction to Singularity Containers on HPC. Artículo de DOD HPC (https://centers.hpc.mil/users/singularity.html). Consultado el 12 de mayo del 2021.

[152] Safak Ulusoy. How Docker Container Networking Works - Mimic It Using Linux Network Namespaces. Artículo de Dev (https://dev.to/polarbit/how-docker-container-networking-works-mimic-it-using-linux-network-namespaces-9mj). Consultado el 12 de mayo del 2021.

[153] Insu Jang. Container Runtime. Github Pages de insujang (`https://insujang.github.io/2019-10-31/container-runtime/`). Consultado el 13 de mayo del 2021.

[154] Docker overview. Artículo de Docker Docs (`https://docs.docker.com/get-started/overview/`). Consultado el 13 de mayo del 2021.

[155] Nick Janetakis. Understanding How the Docker Daemon and Docker CLI Work Together. Artículo del blog nickjanetakis (`https://nickjanetakis.com/blog/understanding-how-the-docker-daemon-and-docker-cli-work-together`). Consultado el 13 de mayo del 2021.

[156] Tiffany Jernigan. Docker 1.1 et plus: Engine is now built on runC and containerd. Artículo de Medium (`https://medium.com/tiffanyfay/docker-1-11-et-plus-engine-is-now-built-on-runc-and-containerd-a6d06d7e80ef`). Consultado el 13 de mayo del 2021.

[157] Matthias Leitner. Docker - Its components and the OCI. Github Pages de accenture (`https://accenture.github.io/blog/2021/03/18/docker-components-and-oci.html`). Consultado el 13 de mayo del 2021.

[158] What is a Container? Página de la web de Docker (`https://www.docker.com/resources/what-container`). Consultado el 13 de mayo del 2021.

[159] Sofija Simic. Docker Image vs Container: The Major Differences. Artículo de phoenixNAP (`https://phoenixnap.com/kb/docker-image-vs-container`). Consultado el 13 de mayo del 2021.

[160] What is a container registry? Artículo de Red Hat (`https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry`). Consultado el 13 de mayo del 2021.

[161] Página principal de Docker Hub (`https://hub.docker.com/`). Consultado el 13 de mayo del 2021.

[162] Abhinav Nath Gupta. Desmitifying Kubernetes. Artículo de OpenSource for U (`https://www.opensourceforu.com/2019/11/demystifying-kubernetes/`). Consultado el 14 de mayo del 2021.

[163] What is container orchestration? Artículo de Red Hat (`https://www.redhat.com/en/topics/containers/what-is-container-orchestration`). Consultado el 14 de mayo del 2021.

[164] Isaac Eldridge. What is Container Orchestration? Artículo de New Relic (`https://newrelic.com/blog/best-practices/container-orchestration-explained`). Consultado 14 de mayo del 2021.

[165] Introduction to Kubernetes architecture. Artículo de Red Hat (https://www.redhat.com/en/topics/containers/kubernetes-architecture). Consultado el 14 de mayo del 2021.

[166] Kubernetes Architecture. Artículo de Aqua Cloud Native Academy (https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-architecture/). Consultado el 14 de mayo del 2021.

[167] Pixiake, Feynman & Sherlock. Dockershim Deprecation: Is Docker Truly out of Game? Artículo de Kubesphere (https://kubesphere.io/blogs/dockershim-out-of-kubernetes/). Consultado el 14 de mayo del 2021.

[168] Dockershim Deprecation FAQ. Artículo del blog de Kubernetes (https://kubernetes.io/blog/2020/12/02/dockershim-faq/). Consultado el 14 de mayo del 2021.

[169] Kunal Kushwaha. How Container Runtimes matter in Kubernetes? NTT OSS Center, 2017. Diapositivas disponibles en (https://events19.linuxfoundation.org/wp-content/uploads/2017/11/How-Container-Runtime-Matters-in-Kubernetes_-OSS-Kunal-Kushwaha.pdf). Consultado el 14 de mayo del 2021.

[170] Ahmet Alp Balkan. Choosing an init process for multi-process containers. Artículo del blog ahmet (https://ahmet.im/blog/minimal-init-process-for-containers/). Consultado el 27 de mayo del 2021.

[171] Mathew Palmer. How does Kubernetes use etcd? Artículo de la web mathewpalmer (https://matthewpalmer.net/kubernetes-app-developer/articles/how-does-kubernetes-use-etcd.html). Consultado el 28 de mayo del 2021.

[172] Ignacio Castelo. Aproximación a contenedores. Github Pages de casteloig (https://casteloig.github.io/Aproximacion-Contenedores/)