Trabajo Final Integrador

Alumno: Ing. Fabián Ariel Hidalgo (fabianhdlg@gmail.com)

Especialización en Redes de Datos
Universidad Tecnológica Nacional
Modalidad a distancia

Director: Esp. Ing. Gaston Cangemi

Infraestructura de Soporte para Aplicaciones con Arquitectura Orientada a Microservicios

Objetivo

Con el objetivo de obtener la titulación de Especialista en Redes de Datos, se presenta una propuesta de trabajo final en la temática "Sistemas Distribuidos", bajo el título de "**Infraestructura de soporte para aplicaciones con Arquitectura orientada a Microservicios".**

Justificación del tema

La arquitectura de microservicios ha ganado popularidad en años recientes y por una buena razón: su correcta implementación puede producir numerosas ventajas en nuestra moderna era de la computación en la nube. Las aplicaciones con arquitectura basada en microservicios se apoyan sobre una infraestructura denominada características de núcleo (core features). A lo largo de este trabajo se abordará el estudio de algunas implementaciones de infraestructura denominadas frameworks arquitecturales, implementaciones que han marcado el rumbo de avance en esta área, todo esto desde una perspectiva de administración de infraestructura. Es imperioso para los administradores de sistemas conocer la infraestructura sobre la que se apoya el modelo y las herramientas que permiten administrarla, ya que como se mencionó hay una tendencia fuerte hacia este modelo de arquitectura.

Resumen

El presente trabajo pretende abordar el estudio de la infraestructura de soporte de un tipo de aplicaciones que entran en la categoría de sistemas distribuidos, la clase de aplicaciones a las que se hace referencia, son las denominadas aplicaciones con arquitectura basada en microservicios. Lo que se expone en este trabajo inicia con un conjunto de definiciones que nos permiten visualizar el motor que impulsó el desarrollo de las tecnologías que se examinarán y sientan las bases del marco conceptual que será objeto de referencia a lo largo de todo este documento. Luego se analizaran una serie de implementaciones que si bien no representan estándares para el tipo de infraestructura que es objeto de estudio, son plataformas que sentaron las bases de productos que hoy están en el mercado como alternativas de soporte. La elección arbitraria de dichas plataformas responde a que se pretende acercar, mediante este estudio, el análisis de plataformas que fueron influyentes en el estado actual de las tecnologías disponible en esta área.

Summary

This work seeks to tackle study over the applications support infrastructure which fall into the category of distributed systems, the class of applications referred to are the so-called microservice-based applications. What is exposed in this work begin with a set definitions that allow to visualizing the engine that drove the development of technologies that will be examined and lay the foundations of the framework that will be object of reference throughout this document. Then a series of implementations will be analyzed that, although they do not represent standards for the type of infrastructure that is the object of study, are platforms that laid the foundations for products that are currently on the market as support alternatives. The arbitrary choice of these platforms responds to the intention to bring closer, through this study, of the analysis of platforms that were influential in the current state of the technologies available in this area.

Índice Temático

Sistemas Distribuidos: Arquitecturas, Implementaciones e Infraestructura

Sistemas Distribuidos: Conceptos y Arquitecturas

Examinando las definiciones de Sistema Distribuido en la bibliografía, el autor cita las siguientes las cuales son fundamentales para introducir al tema que interesa y además servirá como referencia a lo largo de éste trabajo:

"Un sistema distribuido es una colección de elementos de cómputo autónomos que aparecen a sus usuarios como un sistema coherente simple."

"...uno en el cual los componentes de hardware o software localizados en una red de computadoras se comunican y coordinan sus acciones únicamente a través del intercambio de mensajes."

La primera definición acentúa dos características clave de los sistemas distribuidos. La primera es que un sistema distribuido es una colección de elementos computacionales cada uno pudiéndose comportar independientemente de los demás. Al elemento computacional se lo denomina nodo, el cual puede ser un dispositivo de hardware o un proceso de software. La segunda característica es que los usuarios creen que están tratando con un solo sistema. Esto significa que de algún modo u otro los nodos autónomos necesitan colaborar y es aquí donde radica el núcleo del desarrollo de un sistema distribuido.

Sistema de elementos de cómputos autónomos

Un principio fundamental que caracteriza a los sistemas distribuidos es que cada uno de sus elementos actúan independientemente uno de otro, sin embargo de esta definición se infiere que son independientes pero no se ignoran unos a otros ya que si así fuera no tendría sentido colocarlos en un mismo sistema distribuido. En realidad los nodos están programados para alcanzar objetivos comunes y esto lo logran mediante el intercambio de mensajes unos con otros.

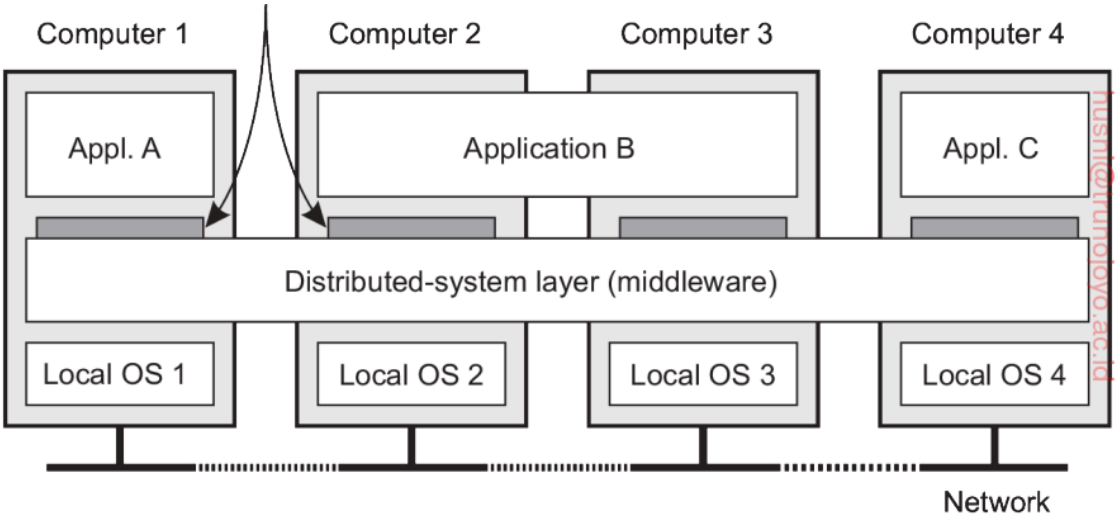
En cuanto a la organización de la colección de nodos que conforman el sistema distribuido, la práctica muestra que un sistema distribuido está frecuentemente organizado como una red superpuesta. Esto significa que un nodo es típicamente un proceso de software equipado con una lista de otros procesos a los cuales puede enviar mensajes directamente.

Sistema coherente único

Que el sistema distribuido aparezca como un único sistema significa que los usuarios finales no deben ni siquiera notar que están tratando con el hecho que procesos, datos y control están dispersos a través de la red de computadoras. Específicamente en un único sistema coherente la colección de nodos como un todo funciona por igual independientemente donde, cuando y como la interacción entre un usuario y el sistema tiene lugar. Un concepto que se desprende de esta característica es la denominado "transparencia de la distribución".

Middleware en sistemas distribuidos

El término middleware aplica a una capa de software que provee una abstracción de programación como así también enmascarado de la heterogeneidad redes subyacentes, hardware, sistemas operativos y lenguajes de programación. Además de resolver problemas de heterogeneidad, el middleware provee un modelo computacional uniforme para el uso por parte de programadores de servidores y aplicaciones distribuidas.



Transparencia de la distribución

Una analogía que resulta muy conveniente es la siguiente:

"el middleware es a un sistema distribuido lo que el sistema operativo es una computadora".

Un administrador de recursos que ofrece sus aplicaciones desplegar y compartir recursos eficientemente

La principal diferencia con su equivalente en sistemas operativos es que los servicios de middleware se ofrecen en un entorno de red. Notemos también que la mayoría de los servicios son usados por muchas aplicaciones. En ese sentido, middleware puede ser también visto como un contenedor de componentes usados comúnmente y funciones que ahora ya no tienen que ser implementadas a través de aplicaciones separadas.

Modelos Arquitecturales Distribuidos

La arquitectura de un sistema es su estructura en términos de componentes especificados de manera separada y sus relaciones. Se examinarán brevemente los modelos arquitecturales empleados en los sistemas distribuidos para proveer un marco de referencia para la arquitectura especifica que el autor trata en este trabajo.

Los bloques de construcción fundamentales de los sistemas distribuidos presentes en cualquier arquitectura son:

Entidades y paradigmas de comunicación: resulta útil abordar las entidades que se comunican desde dos perspectivas: la orientada a sistemas y la orientada al problema.

Desde una perspectiva de sistemas la entidades que se comunican son típicamente procesos donde la comunicación es soportada por el paradigma de comunicación interprocesos.

Desde un perspectiva orientada al problema se ha propuesto la siguiente entidades:

- **Objetos: en las aproximaciones distribuidas basadas en objetos, una computación consiste de un número de objetos que interactúan representando las unidades naturales de descomposición para el dominio del problema dado.**

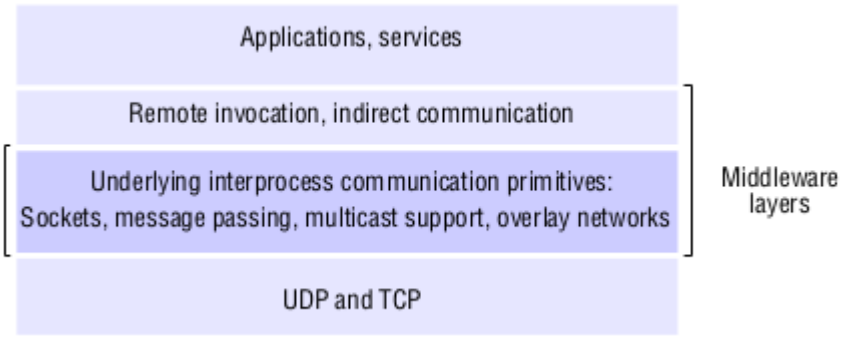
- **Componentes: al igual que los objetos representan unidades de negocio y especifican interfaces para ser accedidos, la diferencia clave radica en que los componentes también especifican los supuestos que hacen en términos de dependencias de otros componentes**

- **Servicios Web: servicios web es el tercer paradigma importante para el desarrollo de sistemas distribuidos. Los servicios web están estrechamente relacionados a los objetos y componentes en que encapsulan el comportamiento y son accedidos a través de interfaces, sin**

embargo los servicios web están intrínsecamente relacionados a la web a través del uso de estándares web para representar y descubrir servicios.

Respecto a cómo las entidades se comunican en un sistema distribuido, examinamos tres paradigmas de comunicación:

- **Comunicación interproceso: se refiere al soporte de bajo nivel para la comunicación entre procesos en sistemas distribuidos, lo que incluye las primitivas para el intercambio de mensajes, el acceso directo a la API ofrecida por los protocolos de Internet y el soporte para comunicación multicast tan necesaria en los sistemas distribuidos**



- **Invocación remota: es el paradigma de comunicación más comúnmente usado en sistemas distribuidos cubriendo un rango de técnicas basadas en intercambio de dos vías entre las entidades que se comunican en el sistema y resultando en el llamado de una operación, procedimiento o método remoto**
  - **Protocolo Solicitud Respuesta: es un patrón impuesto por el servicio de intercambio de mensajes de la capa subyacente para soportar la comunicación de tipo cliente-servidor. Consiste en el intercambio de mensajes de a pares, uno desde el cliente al servidor especificando un código de operación y parámetros en un array y otro de regreso del servidor al cliente con la respuesta en un array.**
  - **Llamada a procedimiento remoto: procedimientos en procesos en computadoras remotas pueden ser llamados como si fueran procedimientos en el espacio local. La capa RPC subyacente soporta esto.**
  - **Invocación de método remoto: métodos de objetos remotos pueden ser llamados**

por objetos. La capa RMI subyacente soporta esto.

- **Comunicación Indirecta: las anteriores técnicas de comunicación tienen en común que la comunicación implica una relación de dos vías entre emisor y receptor y donde emisores dirigen mensajes o invocaciones explícitamente a receptores asociados. Los receptores generalmente también están al tanto de la identidad de los emisores y en la mayoría de los casos ambas partes deben existir al mismo tiempo. Bajo el paradigma de comunicación indirecta, la comunicación se lleva a cabo a través de una tercera parte permitiendo un alto grado de desacoplamiento tanto en tiempo como en espacio. Dentro de este paradigma entran las siguientes tecnicas de comunicacion:**
  - **Comunicación grupal**
  - **Sistemas Publicador - Suscriptor**
  - **Colas de Mensajes**
  - **Memoria compartida distribuida**

<u>Roles y Responsabilidades</u>: en un sistema distribuido las entidades toman roles y esos roles determinan la arquitectura a ser adoptada. Los estilos arquitecturales que tienen lugar en función de los roles son:

- **Cliente - Servidor: las entidades toman el rol de cliente o servidor. En particular, las entidades clientes interactúan con entidades servidoras individuales en host separados. Note que una entidad puede tener los roles de cliente y servidor por ejemplo el caso de un motor de búsquedas.**
- **Peer-to-Peer: en esta arquitectura la mayoría de las entidades involucradas en una tarea juegan roles similares. No hay distinción entre entidades clientes y servidoras, las entidades actúan como pares. En términos prácticos todas las entidades que participan corren el mismo programa y ofrecen las mismas interfaces unas a otras.**

<u>Ubicación</u>: se refiere a cómo las entidades se mapean en la infraestructura distribuida física. La estrategias de ubicación son:

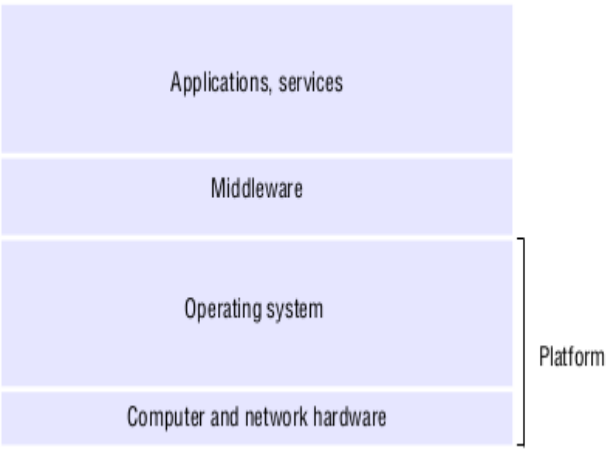
- **Mapeo de servicios a múltiples servidores**
- **Cache**
- **Código Móvil**
- **Agentes Móviles**

Patrones Arquitecturales

Los patrones arquitecturales se construyen sobre elementos arquitecturales más primitivos y proveen estructuras compuestas que se ha mostrado que trabajan bien en determinadas circunstancias. Algunos patrones arquitecturales clave son:

- **Layering: bajo esta arquitectura un sistema complejo es particionado en un número de capas, con una capa dada haciendo uso de los servicios ofrecidos por las capas inferiores. Una capa dada por lo tanto ofrece una abstracción de software, con las capas más altas despreocupadas por los detalles de implementación de las capas inferiores. En términos de sistemas distribuidos, esto iguala a una organización vertical de servicios dentro de capas de servicio. Un servicio distribuido puede ser provisto por uno o mas procesos servidores que interactúan unos con otros y con los procesos cliente para mantener una vista consistente de todo el sistema de recursos de servicio. Es importante en este tema introducir el concepto de plataforma y middleware.**

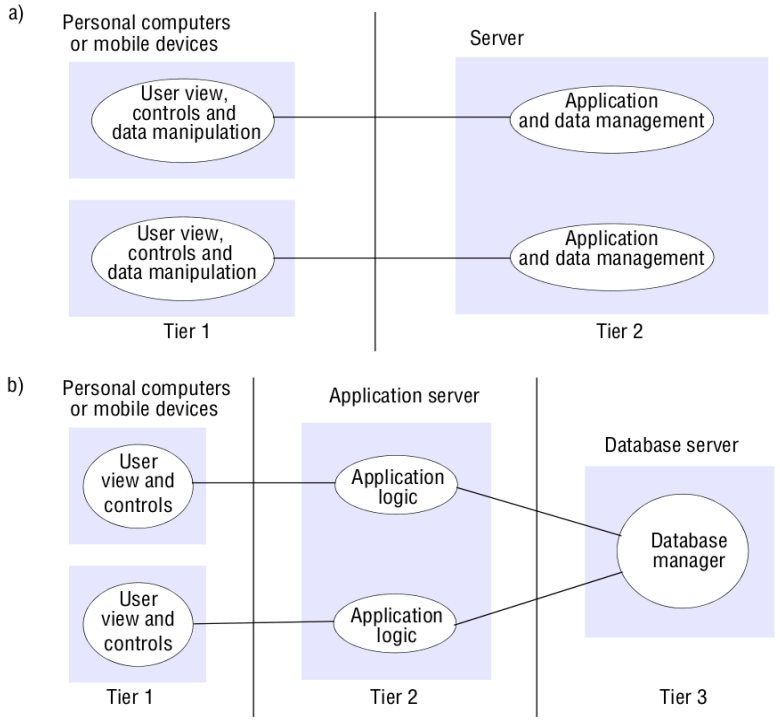
  - **Plataforma: en un sistema distribuido consiste de las capas más bajas de hardware y software. Esas capas de bajo nivel proveen servicios a las capas encima de ellas, los cuales son implementados independientemente en cada computadora llevando la interfaz de programación de sistemas hasta un nivel que facilita la comunicación y coordinación entre procesos.**



  - **Middleware: se interesa en la provisión de útiles bloques de construcción para el armado de componentes de software que puedan trabajar con algún otro en un sistema distribuido. En particular eleva el nivel de las actividades de comunicación de programas de aplicación a través del**

soporte de abstracciones tales como RMI, comunicación entre un grupo de procesos, particionado, notificación de eventos, ubicación y recuperación de objetos de datos compartidos entre computadoras que cooperan, replicación de objetos de datos compartidos.

- **Arquitectura Multinivel: esta arquitectura es complementaria a la arquitectura por capas, se ocupa de organizar funcionalmente una capa dada y ubicar esta funcionalidad en servidores apropiados.**



Arquitectura Orientada a Servicios: De SOA a Microservicios

SOA (Services Oriented Architecture)

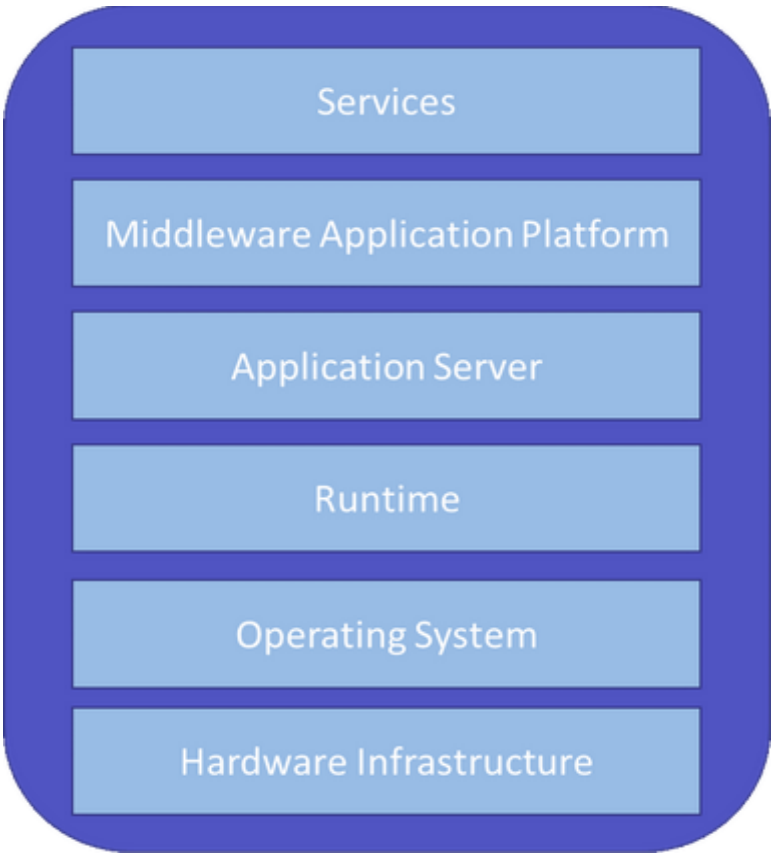
Luego de introducir las características clave de los sistemas distribuidos en general y los patrones arquitecturales que rigen su diseño se abordará el estudio del modelo arquitectural cuyo aspecto particular, a saber la infraestructura que sustenta su implementación, es objeto de estudio de este trabajo.

El autor inició este trabajo mediante una introducción a los sistemas distribuidos sencillamente porque la arquitectura orientada a microservicios es un tipo de arquitectura distribuida. Microservicios es un patrón de

arquitectura que estructura la aplicación como una composición de servicios débilmente acoplados que están no sólo lógicamente separados sino que también están físicamente separados en tiempo de ejecución. Los microservicios son componentes (entidades de acuerdo a la definición previa de elementos arquitecturales) autónomos ligeros de granularidad fina.

Resulta útil para iniciar el estudio del modelo arquitectural analizar su origen y para ello el autor se ubica en la arquitectura orientada a servicios o SOA. SOA es un estilo de diseño de software donde una aplicación es construida como un conjunto de servicios. Un servicio es una unidad discreta de funcionalidad que puede ser accedida remotamente, a través de una interfaz, y ejecutada y actualizada independientemente.

SOA es la respuesta de diseño (y no es la única es decir existen otras) a un modelo de componentes (un componente puede terminar siendo un paquete de software, un servicio web, un módulo, etc.). Los diferentes servicios pueden ser usados en conjunto para proveer la funcionalidad de una gran aplicación de software. La arquitectura orientada a servicios integra componentes de software desplegados, mantenidos separadamente y distribuidos. SOA promueve el débil acoplamiento entre servicios. SOA separa funciones en distintos servicios los cuales los programadores hacen accesible a través de la red de modo de permitir a los distintos usuarios combinarlos y reutilizarlos en la producción de aplicaciones. Estos servicios y sus correspondientes consumidores se comunican entre ellos a través del paso de datos mediante un formato compartido bien definido o mediante la coordinación de actividades entre dos o más servicios.



Arquitectura de SOA

Algunas características de SOA que interesa enfatizar son:

- **Basada en el principio de que cada servicio publica una gran función de negocio (grano grueso)**
- **Los servicios son desplegados sobre pesadas plataformas de middleware centrales (servidores de aplicaciones) donde todos ellos comparten la misma infraestructura de cómputo y almacenamiento**
- **Apropiado para integración de sistemas empresariales**

Algunas implementaciones de SOA son:

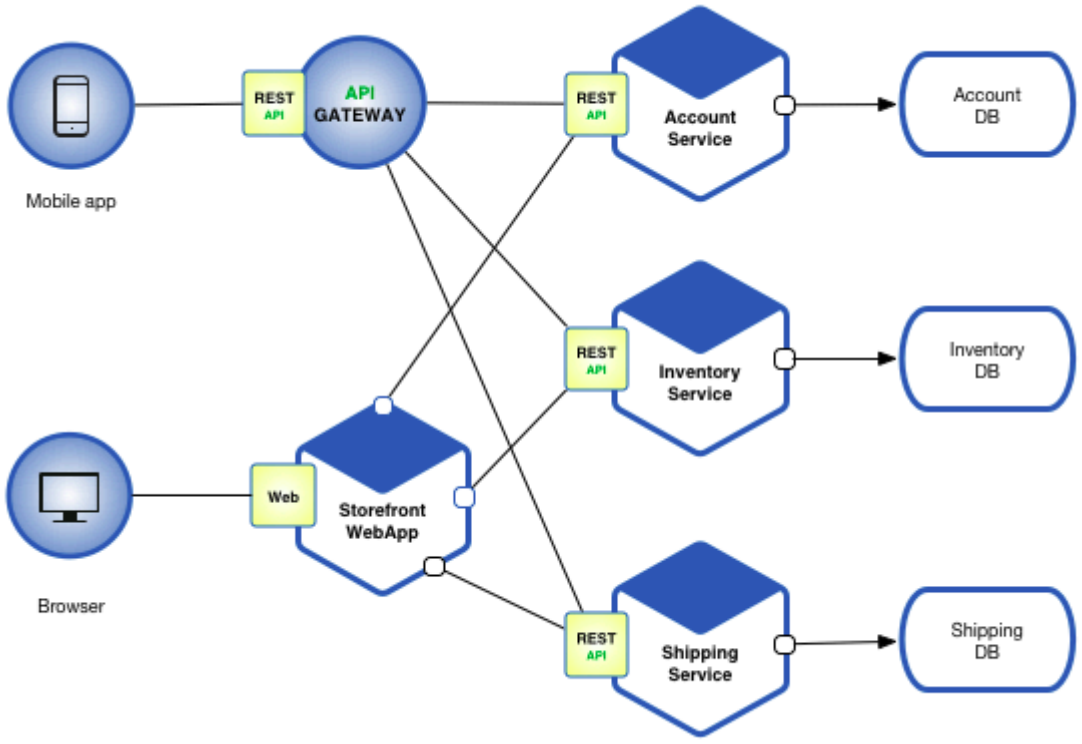
- **Web Services**
- **JavaSpaces**
- **Jini**

MSA (Microservices Architecture)

Microservicios es un patrón de arquitectura que estructura una aplicación como una composición de servicios débilmente acoplados que no solo están lógicamente separados si no que también están físicamente separados en tiempo de ejecución. Los microservicios son componentes de software livianos de grano fino. Claro que este patrón arquitectural esta precedido por una estrategia de modelado de dominio que permite el uso de esta modelo arquitectural en la etapa de diseño, algunas estrategias son:

- **Descomposición por capacidades de negocio**
- **Descomposición por subdominios**
- **Descomposición por casos de uso y definición de servicios que son responsables de acciones particulares**

El siguiente gráfico muestra una aplicación e-commerce con una posible descomposición en microservicios. La aplicación toma pedidos de clientes, verifica inventario y crédito disponible y realiza el envío. Así la aplicación está formada por un conjunto de componentes entre los que estan componentes de interfaz de usuario y componentes de backend formado por servicios de backend como son servicio de cuenta, servicio de inventario y servicio de envío :

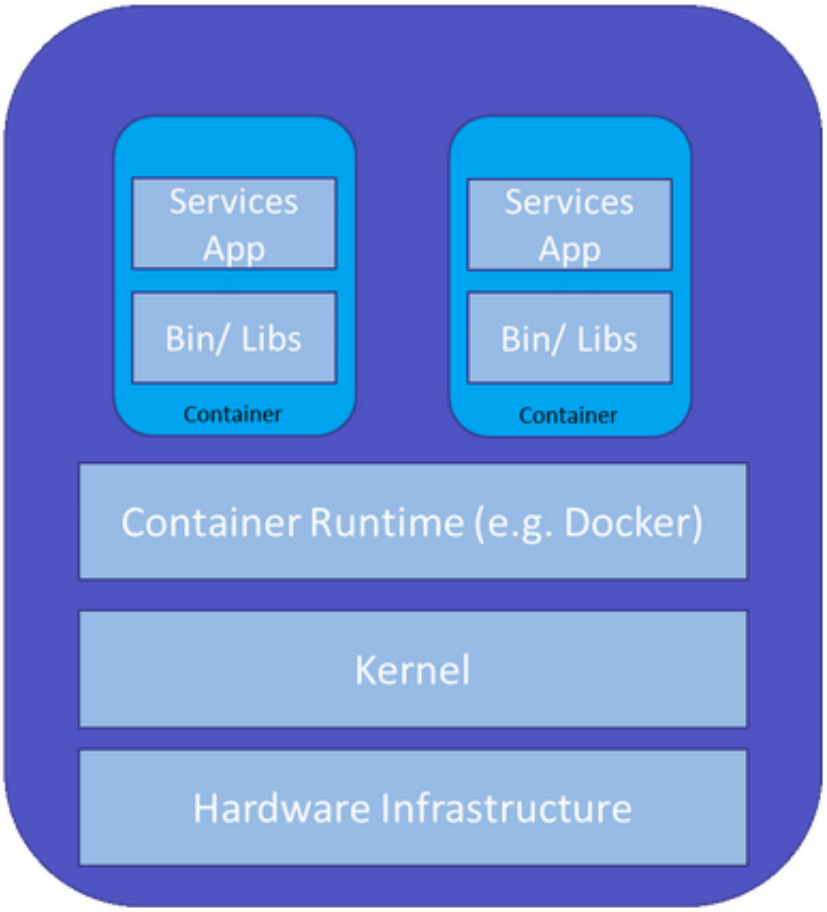


Aplicación bajo arquitectura MSA

Los beneficios de la arquitectura son:

- **Permite la entrega y despliegue de grandes y complejas aplicaciones**
    - **Altamente mantenibles y testeables: cada servicio es relativamente pequeño y así más fácil de entender, cambiar y testear**
    - **Desplegables independientemente**
- **Mejora el aislamiento de fallas**
- **Elimina cualquier compromiso de largo plazo a una tecnología de stack. Cuando se inicia el desarrollo de un nuevo servicio se está en la libertad de seleccionar la tecnología de stack que se desee.**
- **Facilita las adaptación en tiempo de ejecución de la aplicación: haciendo que las operaciones de update y upgrade puedan llevarse a cabo en tiempo de ejecución sin comprometer la continuidad de la aplicación. La adaptación aquí es reemplazar una instancia de microservicio con otra o crear nuevas réplicas de microservicios.**
- **Capacidad de autoescalado: cada servicio escala independientemente unos de otros mediante el empaquetado en contenedores y el escalado individual bajo demanda**

La siguiente es una vista esquemática de la arquitectura:



Arquitectura MSA

Son evidentes, al comparar la vista esquemática de SOA y MSA, las diferencias desde la óptica de la infraestructura de soporte a entidades de servicio, SOA por estar basado en el principio de que cada servicio expone un gran número de funciones de negocio requiere una gran plataforma middleware de soporte mientras que MSA por estar basado en el principio que cada entidad de servicio realiza una pequeña unidad de trabajo no requiere un soporte de middleware de gran complejidad.

Por primera vez en este trabajo el autor referencia el componente clave de implementación del modelo de microservicios, los contenedores. Esta tecnología es tema del siguiente capítulo donde definitivamente el autor ingresa al mundo de la implementación del modelo arquitectural de microservicios.

Implementación de Microservicios

Soporte de kernel para contenedores

Los contenedores son en la actualidad el núcleo de implementación de sistemas distribuidos, entre ellos, los sistemas cuyo modelo arquitectural es el de microservicios. En esta sección se abordará el estudio de contenedores desde la perspectiva de implementación a nivel sistema operativo y se hará una introducción a las tecnologías de facto existentes en la actualidad que proporcionan un entorno de implementación para aplicaciones basadas en ellos.

El concepto de contenedor

Un contenedor es una unidad en ejecución sobre un entorno que soporta contenedores. Que un entorno soporte contenedor significa que cuenta con los mecanismos a nivel sistema operativo y con el software adicional para lograr poner en ejecución una imagen de aplicación autocontenida, esto es, un paquete que contiene todo lo que necesita una aplicación para ser ejecutada, es decir, código, librerías, frameworks y archivos de configuración. Más adelante se examinara con detalle el soporte a nivel sistema operativo y las tecnologías de software para lograr tanto la ejecución de aplicaciones sobre contenedores como la administración de los mismos.

Breve historia de contenedores

- **1979 chroot en Unix: El concepto de contenedores emerge en 1979, cuando se incorpora en la versión 7 de Unix la llamada al sistema chroot, la cual cambia del directorio raíz de un proceso y sus hijos a una nueva ubicación. Esta llamada al sistema fue el inicio del aislamiento de procesos: aislar el acceso a los archivos para cada proceso.**

- **2000 FreeBSD Jails: cuando un pequeño proveedor de hosting sobre entorno compartido commiteo una funcionalidad luego denominada jails la cual lograba una clara separación entre sus servicios y aquellos que pertenecen a sus clientes con el objeto de mejorar la seguridad y facilitar la administración. Esta funcionalidad se apoyo en chroot y otras tecnologías para lograrlo (rctl,vnet,ect).**

- **2001 Linux VServer: es una funcionalidad del mismo tipo de jail pero sobre Linux. Al igual que jails, VServer puede particionar recursos en un**

sistema de computo (sistema de archivos, direcciones de red, memoria). Se introdujo en 2001 y fue implementada mediante actualización de kernel de Linux (patch de kernel).

- **2004 Contenedores Solaris: fue lanzada la primera beta pública de contenedores Solaris que combina controles de recursos de sistema y separación de límites proporcionado por zonas, lo cual hizo posible características como instantáneas y clonado desde ZFS.**

- **2005 Open VZ: esta es una tecnología de virtualización a nivel sistema operativo la cual utiliza la actualización del kernel de Linux para virtualización, aislamiento y administración de recursos y gestión de puntos de restauración.**

- **2006 Contenedores de proceso: es una implementación más de virtualización a nivel sistema operativo. Fue lanzada por Google en 2006 y como toda tecnología de virtualización de sistema operativo fue diseñada para limitar, contar y aislar el uso de recursos (CPU, I/O,memoria,red) de un conjunto de procesos. Fue renombrada como Control de Grupos (cgroups) un año después e incorporado al kernel de Linux en la versión 2.6.24.**

- **2008 Linux LXC: fue la primera y más completa implementación de administración de contenedores en Linux. Fue implementada en 2008 usando cgroups y namespaces de Linux.**

- **2011 Warden: es un servicio para administración de una colección de contenedores y define un protocolo de solicitud respuesta entre cliente y servidor. Warden puede aislar entornos en cualquier sistema operativo, corriendo como demonio y proporcionando una API para administración de contenedor. Warden inicialmente usaba LXC el cual luego fue reemplazado por una implementación propia.**

- **2013 Docker: cuando Docker emergió en 2013 los contenedores explotaron en popularidad. Al igual que Warden Docker uso inicialmente LXC y luego lo reemplazo por su propia implementación a través de la librería libcontainer. Pero no hay dudas de que Docker se separó del resto ofreciendo un ecosistema entero de administración de contenedores.**

Contenedores y sus ventajas

- **Alineado con DevOps: La contenerización empaqueta una aplicación junto con sus dependencias ambientales lo que asegura que una aplicación desarrollada en un ambiente trabaje en otro. Esto ayuda a los desarrolladores y testers a trabajar colaborativamente en la aplicación lo que representa uno de los puntales de la cultura DevOps.**

- **Múltiples plataformas de nube: los contenedores pueden correr en múltiples plataformas de nube como son GCS (Google Cloud Storage), ECS (Elastic Container Service) de Amazon y DevOps Server de Amazon.**

- **Portable por naturaleza: los contenedores ofrecen fácil portabilidad. Una imagen de contenedor puede ser desplegada fácilmente en un nuevo sistema, que luego se puede compartir en forma de archivo.**

- **Escalabilidad más rápida: Dado que los ambientes son empaquetados en contenedores aislados, se pueden escalar más rápidamente lo cual es muy provechoso para aplicaciones distribuidas.**

- **No se necesita Sistema Operativo separado: en sistemas de máquina virtual, el servidor bare-metal tiene un sistema operativo diferente al de la máquina virtual. Por el contrario en contenedores, la imagen de Docker puede utilizar el kernel del sistema operativo host del servidor físico bare-metal.**

- **Máxima utilización de recursos: La contenerización maximiza la utilización de los recursos computacionales como la memoria y cpu y utiliza bastante menos recursos que las máquinas virtuales.**

- **Actualizaciones de seguridad simplificadas: Dado que los contenedores proporcionan aislamiento de procesos, mantener la seguridad de las aplicaciones se vuelve mucho más conveniente de manejar.**

Ventajas de Contenedores

DevOps y contenedores

La tecnología de contenedores además de proporcionar un modelo de implementación para el modelo arquitectural de microservicios proporciona una serie de ventajas para las etapas de integración, entrega y despliegue. DevOps es una metodología de ingeniería de software que tiene como objetivo unificar el desarrollo de software (Dev) y las operaciones de software (Ops) al llevar o mantener una aplicación en el ambiente productivo. Los sistemas de contenedores permiten empaquetar las dependencias de una aplicación y generar una unidad que podrá ser ejecutada en cualquier entorno (desarrollo, test o producción) que tenga soporte para contenedores, también proporcionan sistemas de administración de imágenes, estas característica de los sistemas contenedores está alineada con la filosofía de DevOps.

Los sistemas de administración de imágenes en ecosistemas de tecnologías de contenedores facilitan a los desarrolladores de software realizar operaciones IT (que antes la llevaba a cabo el operador de infraestructura en otra fase), compartir software y colaborar entre ellos y mejorar la productividad. Además de incentivar a los desarrolladores trabajar juntos logran eliminar el conflicto de diferentes entornos de trabajo que antes afectaba a las aplicaciones. En términos sencillos, los contenedores, al ser de naturaleza dinámica, permiten a los profesionales IT ejecutar pipelines para construir, probar y desplegar aplicaciones sin complejidad al mismo tiempo que se cierra la brecha entre la infraestructura y las distribuciones de sistema operativo. La contenerización empaqueta la aplicación junto con sus

dependencias lo que asegura que una aplicación desarrollada en un entorno trabaje en otro. Esto ayuda a los desarrolladores y testers a trabajar colaborativamente lo cual es exactamente de lo que la cultura DevOps se trata.

Tecnologías de virtualización: Máquinas Virtuales versus Contenedores

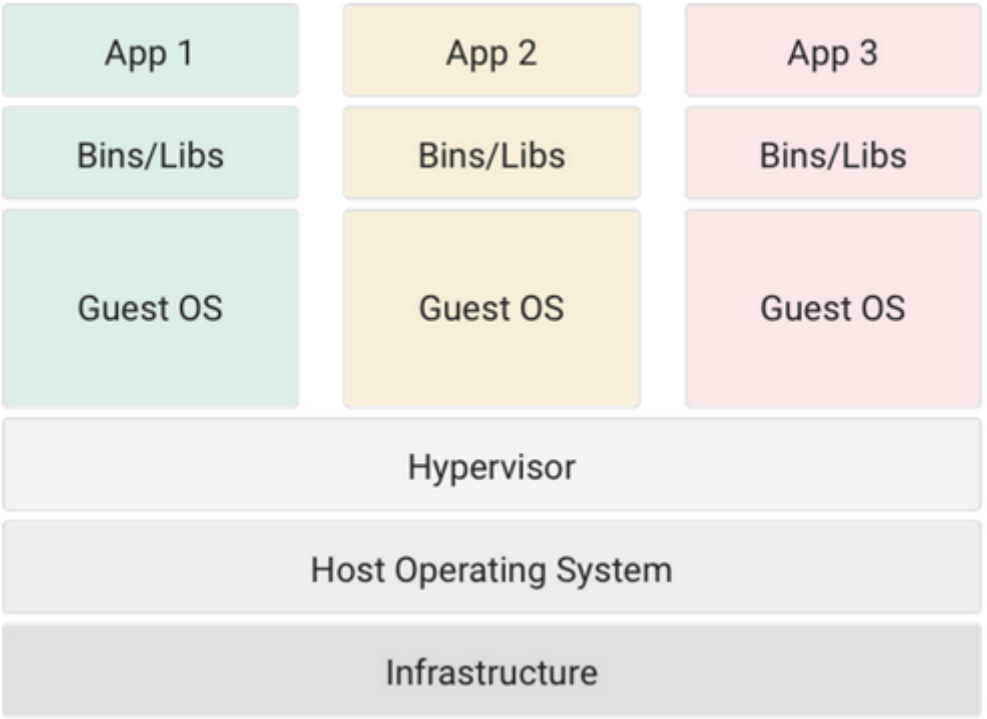
Esta comparación podría también ser llamada Virtualización de Hardware versus Virtualización de Sistema Operativo, lo cual resulta conveniente para la explicación que se pretende abordar, así se define a los contenedores una vez más enfatizando el soporte de capas inferiores, entonces la tecnología de contenedores proporciona una abstracción en capa de aplicación para la ejecución aislada de procesos. Múltiples contenedores pueden ser ejecutados en la misma máquina y compartir el kernel de sistema operativo con otros contenedores, cada uno ejecutándose como proceso aislado en el espacio de usuario. Del otro lado están las máquinas virtuales, las cuales se apoyan en en una capa de virtualización proporcionada por los niveles inferiores de la máquina o por el propio sistema operativo, llamado sistema operativo anfitrión, haciendo posible la ejecución de uno o más sistemas operativos virtualizados, llamado sistema operativo invitado. Tal capa de virtualización es implementada mediante el denominado monitor de máquina virtual o hypervisor quien presenta al invitado una plataforma operativa virtual y administra la ejecución del mismo.

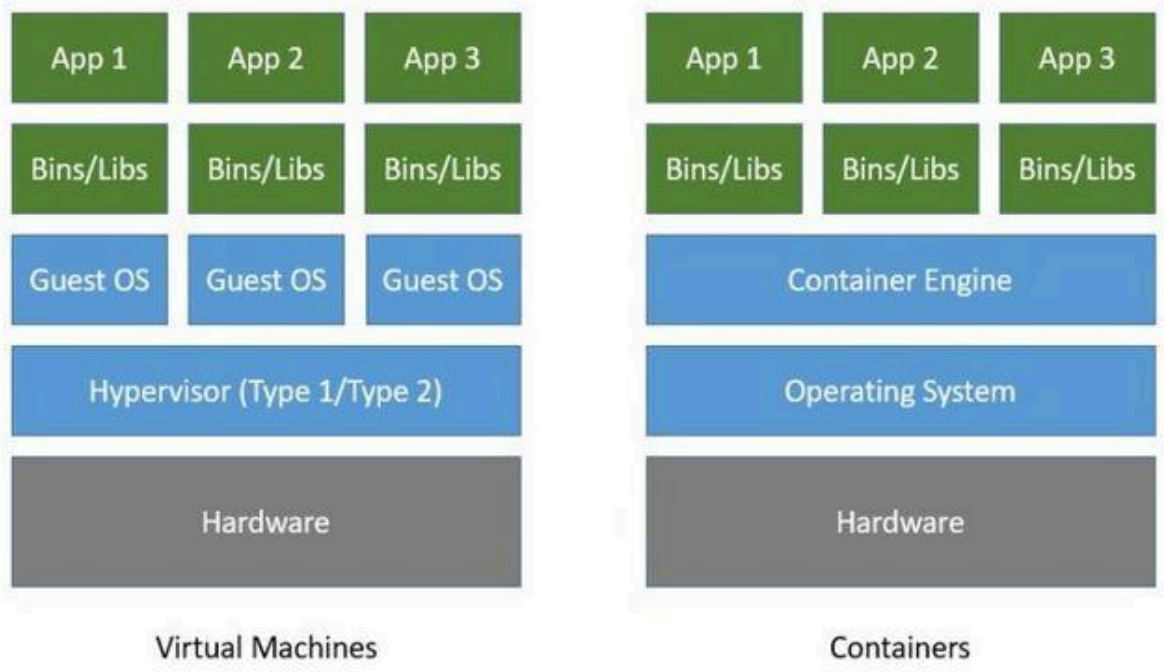
Un análisis comparativo muestra que:

- **La ejecución de un proceso en contenedor puede ser mas rapida que la de un proceso en una maquina maquina virtual: Esto es así porque el código de un proceso en máquina virtual debe bajar a través de la capa de virtualización para recién llegar al nivel de hardware mientras que el código de un proceso de contenedor no pasa por este nivel. El tiempo de inicio de un contenedor también es mucho menor que el de una máquina virtual ya que esta última debe montar un pila de hardware virtual además de cargar el propio kernel invitado.**

- **El espacio ocupado por un contenedor es menor que el ocupado por una máquina virtual: Un contenedor está compuesto por el código de él o los procesos junto con su entorno compartiendo el kernel con otras contenedores mientras que cada instancia de máquina virtual incluye una imagen de kernel.**

- **El escalado horizontal de aplicaciones es más eficiente en una aplicación basada en contenedor ya que el mismo aísla la aplicación y sus dependencias mientras que la máquina virtual lleva consigo además de las dependencias la imagen completa del sistema operativo.**





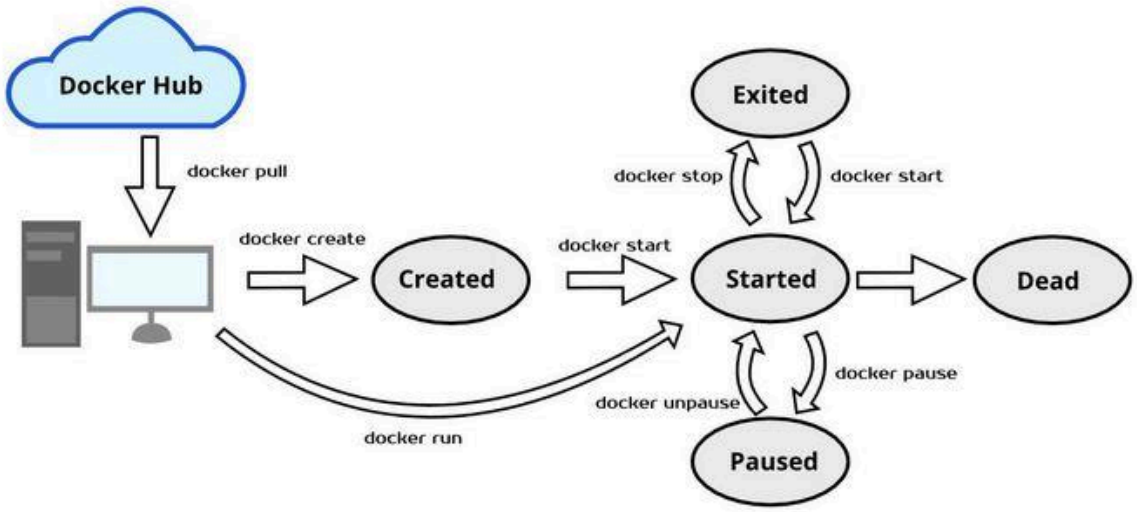
Máquinas Virtuales versus Contenedores

Perspectiva de Implementación de Contenedores

Introducción

En esta sección se profundizará en las tecnologías subyacentes que soportan a los productos actuales de contenedores. El énfasis estará sobre una serie de tecnologías de soporte para virtualización a nivel sistema operativo que es en definitiva el núcleo de implementación de contenedores. En otras palabras se analizará el soporte del sistema operativo para lograr aislamiento de recursos, que como ya se ha mencionado es la clave de la virtualización del sistema operativo. Se sabe que los contenedores requieren de tres categorías de software los cuales están muy relacionadas con su ciclo de vida, estas son:

- **Builder o Constructor de contenedores: se utilizan para la creación de contenedores.**
- **Motor de contenedores: son las tecnologías empleadas para la ejecución de contenedores**
- **Manager u orquestador de contenedores: tecnologías usadas para administrar muchos contenedores.**



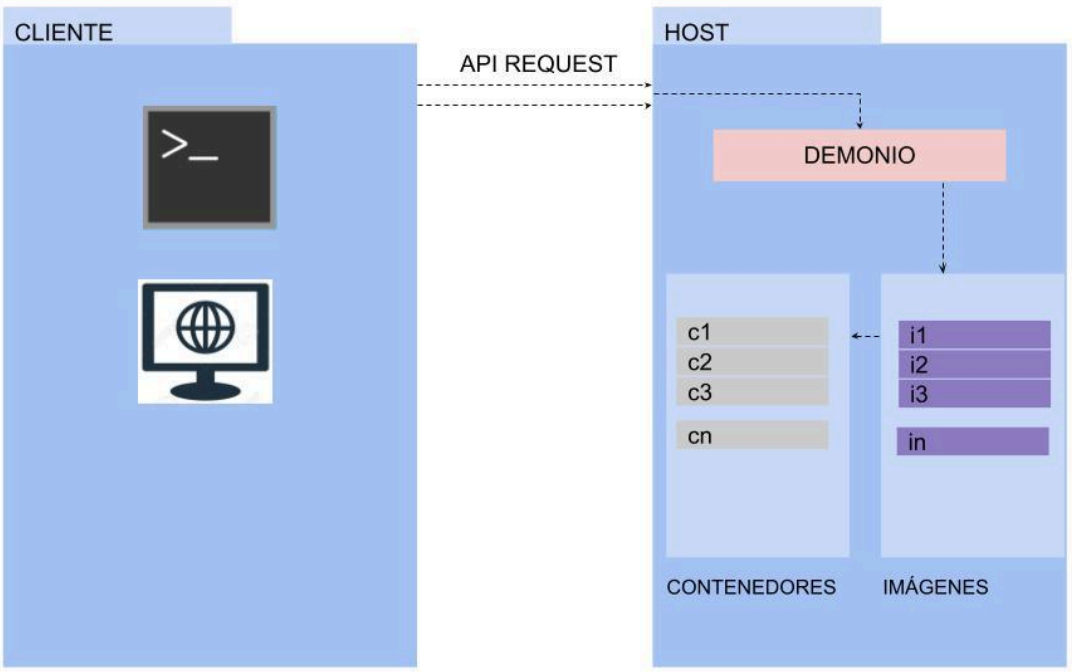
Ciclo de vida de contenedor

Resulta conveniente la anterior clasificación del software asociado con la tecnología de contenedores ya que permite ubicar al lector en el abordaje que se hará de la implementación de contenedores. En esta sección se analizará las herramientas y tecnologías que hacen posible la ejecución de contenedores dentro del ciclo de vida.

Arquitectura de Manager

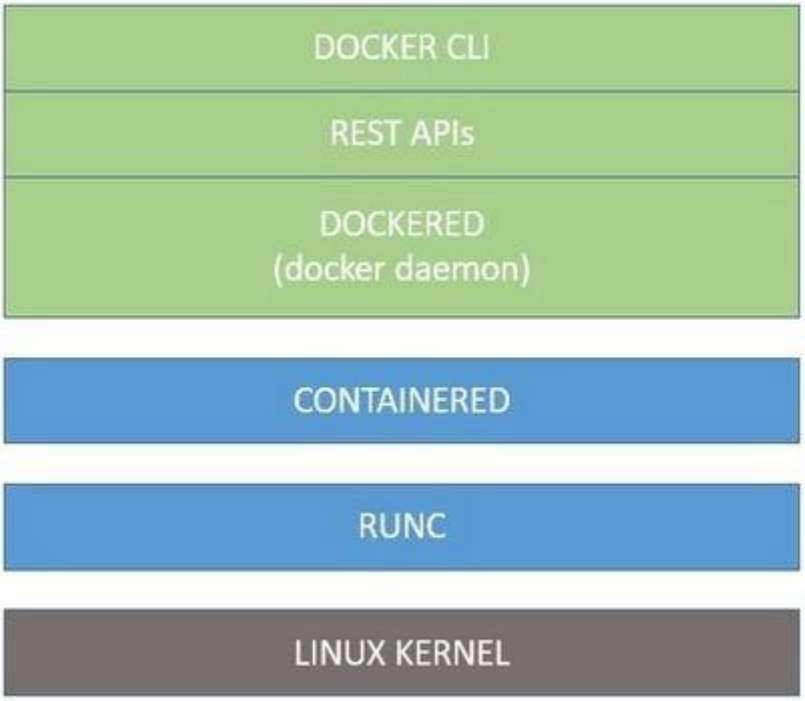
Antes de ingresar en cualquiera de las etapas del ciclo de vida de un contenedor se examinará la arquitectura de los administradores de contenedores. Es importante mencionar que en este estudio de implementación de contenedores el autor ha escogido a Docker como tecnología de referencia permanente por ser una de las más utilizadas mientras que también se hará referencia a LXC siempre que sea conveniente por ser una tecnología pionera en sistemas de administración de contenedores.

En general la arquitectura de un sistema de administración de contenedores es la siguiente:

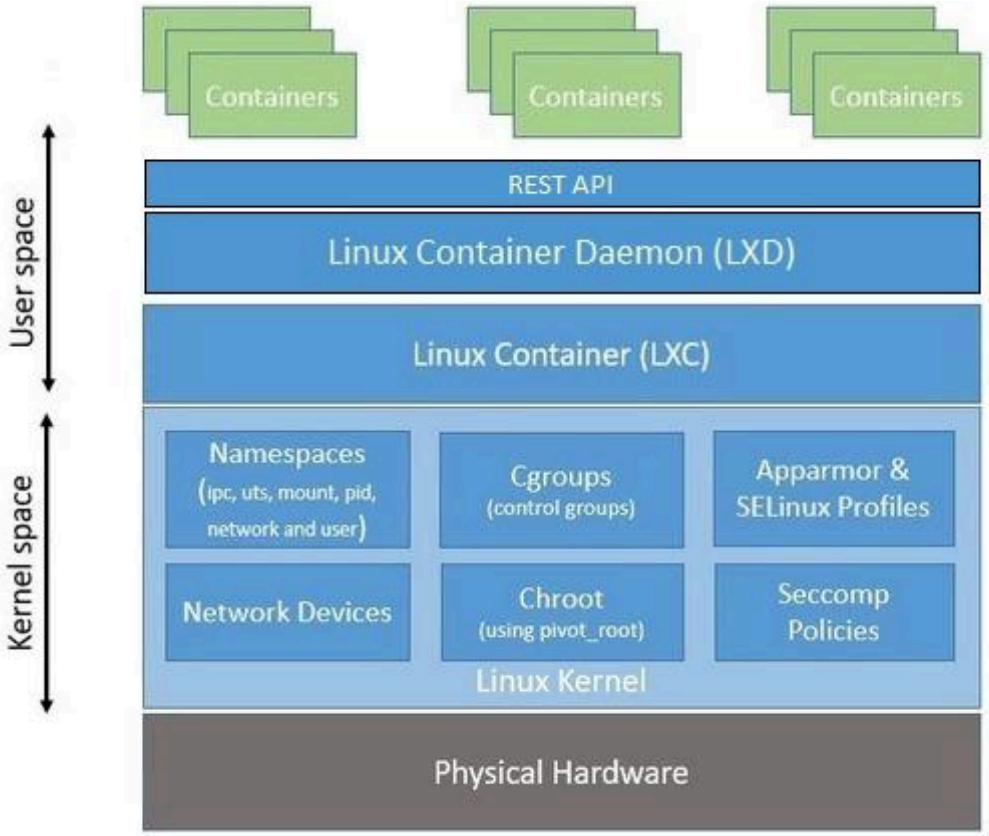


Arquitectura Genérica

El esquema anterior muestra la existencia de un proceso servidor que expone una API formada por el conjunto de operaciones para crear, ejecutar y administrar contenedores. La implementación de estas operaciones está soportada por el kernel del sistema operativo. De las tecnologías que se exponen, ambas fueron diseñadas bajo el modelo cliente servidor de modo tal que existe un proceso, como mencionamos anteriormente, que expone un conjunto de operaciones pero además ese proceso escucha solicitudes en un puerto de comunicaciones. La figuras siguientes muestran la vista arquitectural de Docker y LXC respectivamente.



Arquitectura de Docker



Arquitectura LXC

Soporte del sistema operativo

En esta sección se analizará el conjunto de características que forman parte del kernel de un sistema operativo que hacen posible la virtualización a nivel sistema operativo. En este punto hacer referencia a las primeras tecnologías de contenedores es sumamente útil, por eso vera en detalle el soporte del sistema operativo desde la perspectiva de Linux Containers LXC.

El modelo de procesos

Para iniciar el estudio de la implementación de contenedores es necesario repasar algunos conceptos de sistemas operativos. Se hará énfasis en la abstracción de procesos de un sistema operativo para explicar cómo se llega a la innovación de contenedores.

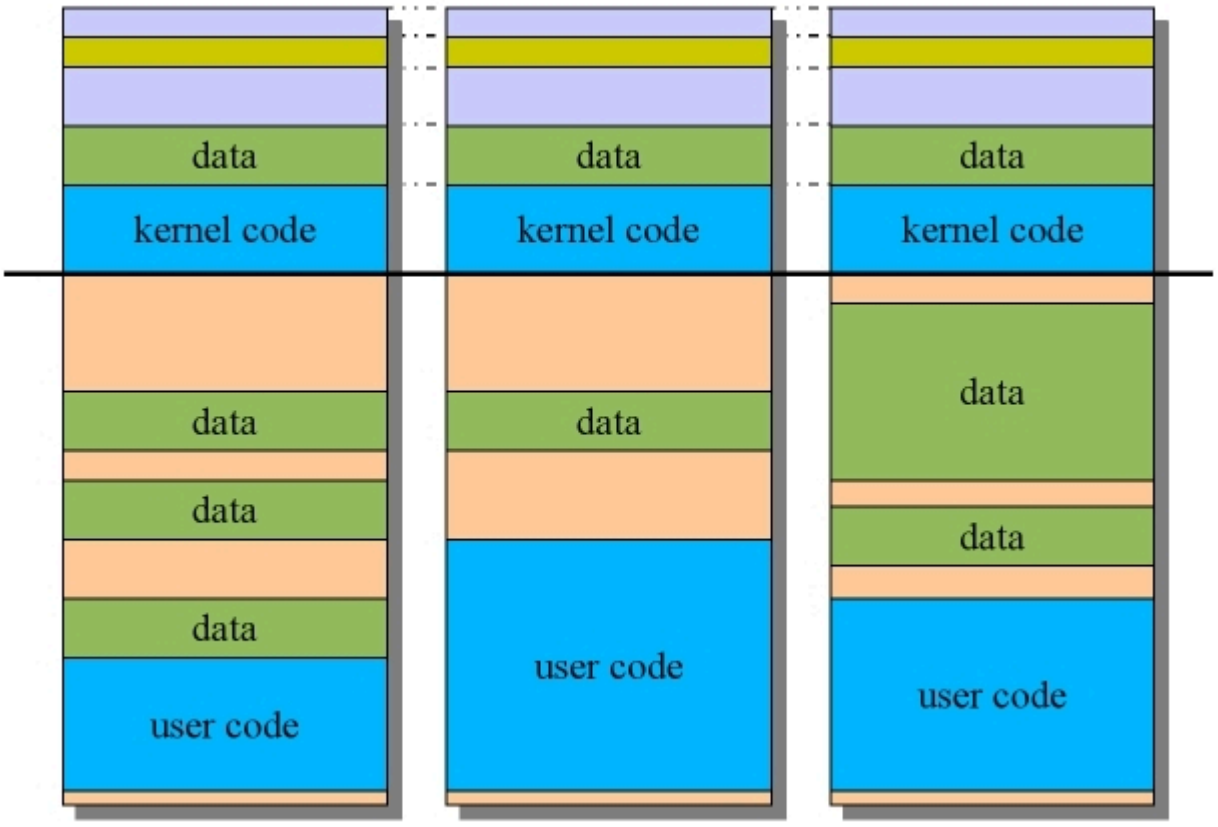
Se sabe que las funciones principales de un sistema operativo son:

- **Proporcionar una representación uniforme y abstracta de los recursos que las aplicaciones pueden solicitar y utilizar.**

- **Controlar la utilización de los recursos.**

Los procesos en un sistema operativo son entidades que se encuentran en un estado de ejecución y formado por:

- **El código de programa**
- **Los datos asociados**
- **Una estructuras de datos que permite al sistema operativo llevar a cabo la función global de supervisor de la utilización de recursos.**



Memoria de Procesos

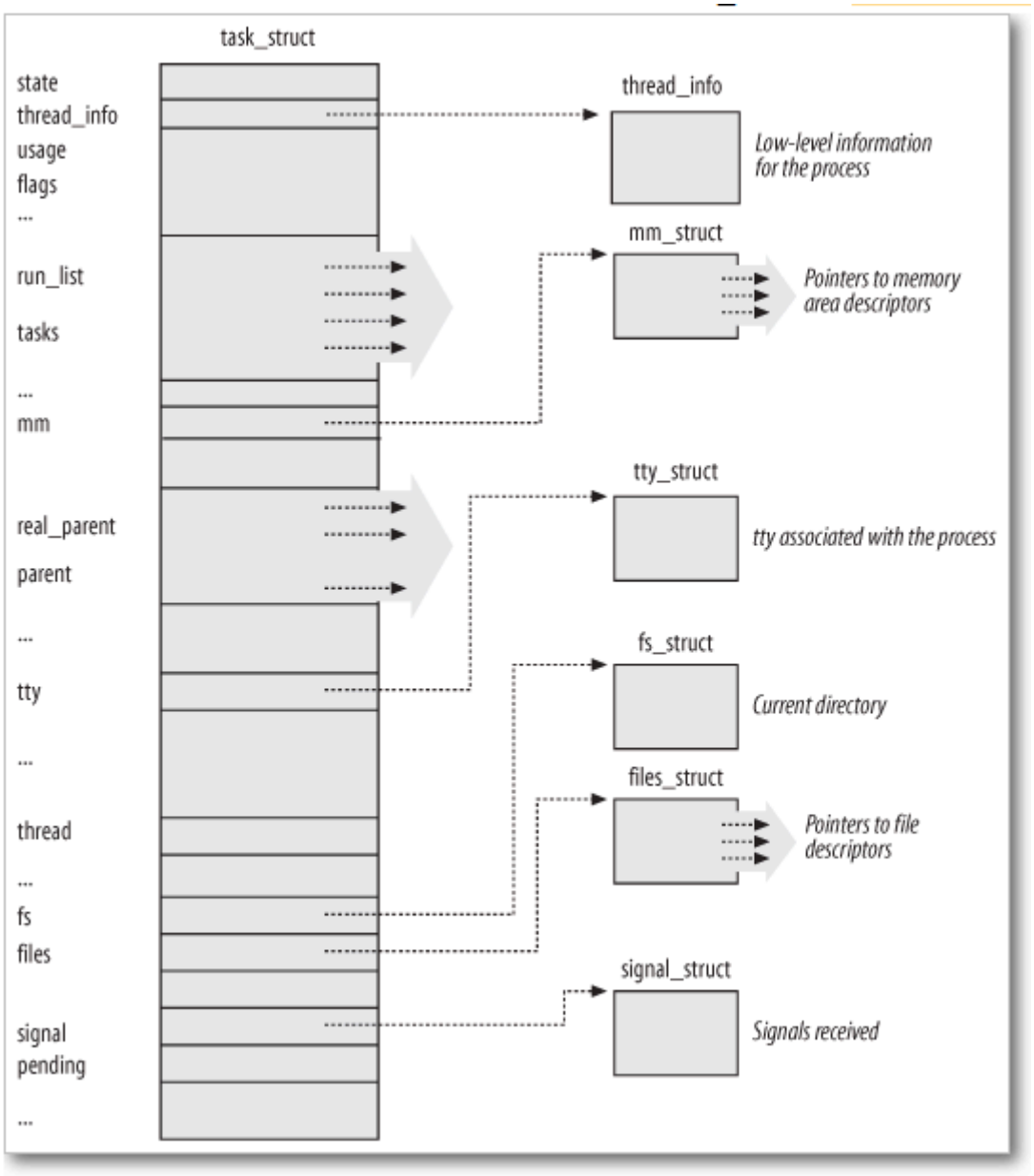
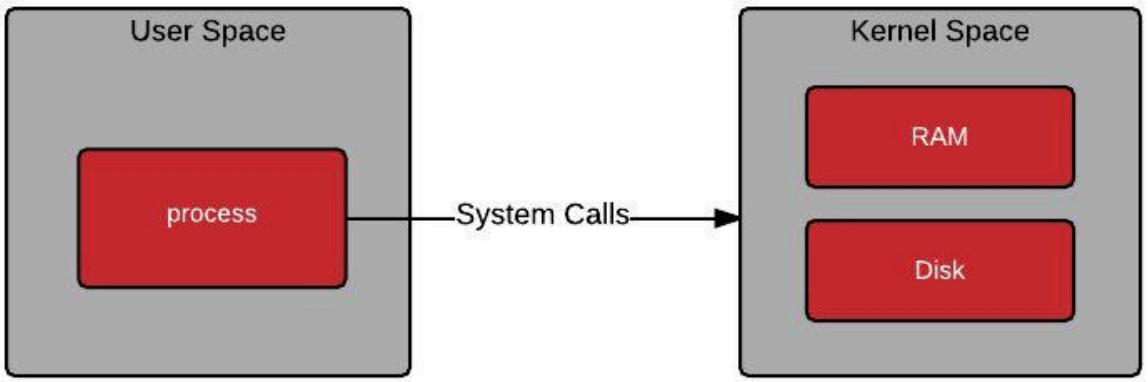


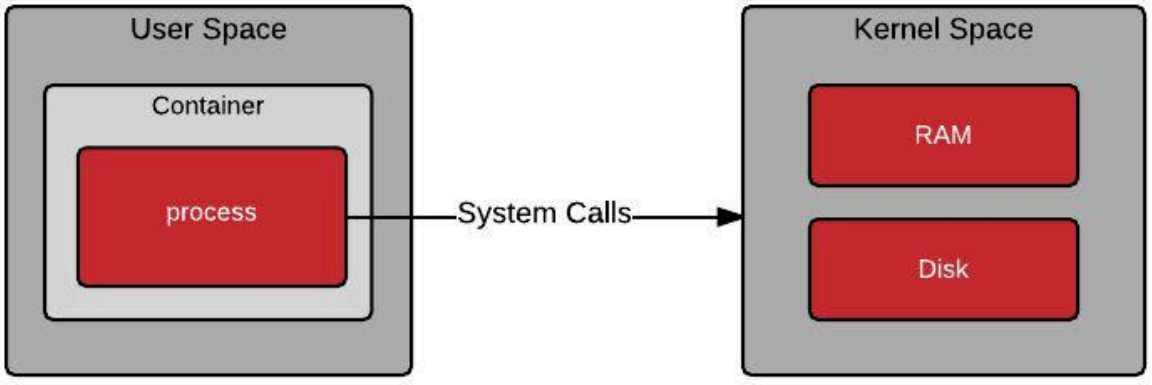
Tabla de Procesos

Es importante notar que por defecto un proceso tiene una mínima separación o aislamiento de recursos. Por ejemplo, se obtendría un error si más de un proceso desean hacer uso de un mismo puerto. Existen dos recursos principales que demuestran la separación de recursos por defecto asociada a los procesos:

- **Memoria**
- **Privilegios restringidos los cuales provienen del usuario que creó el proceso**

Sin embargo la idea de proceso y su implementación antes de los contenedores era solo asignar y controlar el acceso a los recursos en un entorno multitarea. Con la incorporación de la llamada al sistema chroot, es claro que se empieza a pensar en darle a esas entidades que compiten por el uso de recursos computacionales, llamadas procesos, también la posibilidad de ser ejecutados de manera aislada. Tecnologías como jail de BSD, VServer de Linux avanzaron en la implementación de aislamiento de procesos hasta que en el año 2002 se incorpora el soporte en kernel de namespaces de tipo mount agregando en versiones sucesivas otros tipos de namespaces y en el año 2016 se incorporó en el kernel de linux la llamada al sistema cgroups, una verdadera interfaz unificada para diferentes casos de uso en lo que se refiere a control de acceso recursos a un proceso o grupo procesos, desde el control de un solo proceso a un recurso hasta el aislamiento total de un grupo de procesos.



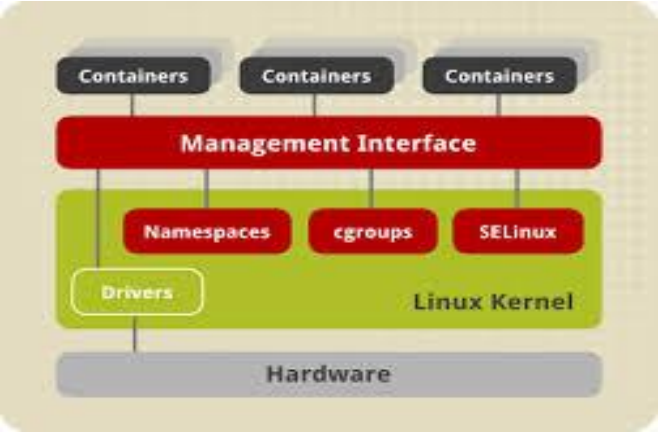


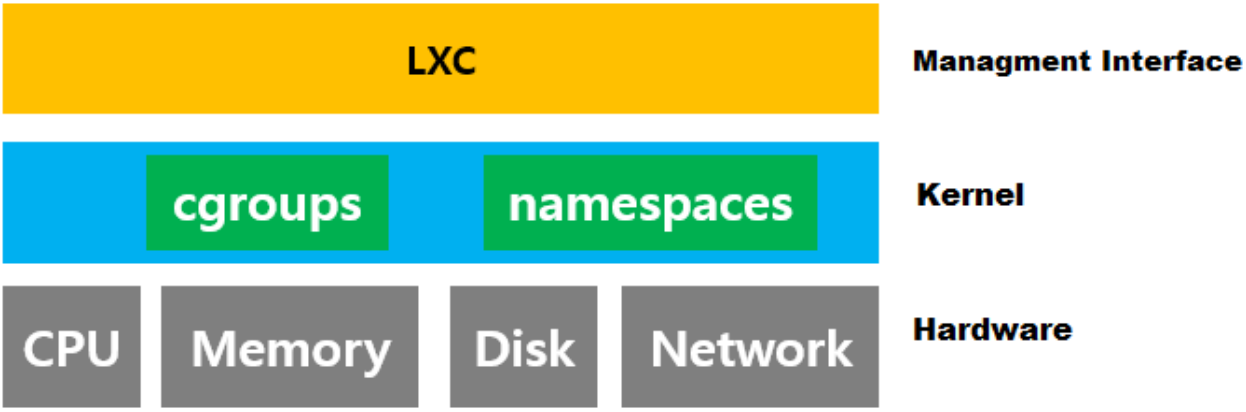
Procesos y Contenedores

El núcleo de implementación de virtualización de sistema operativo

En las siguientes secciones se examinarán las dos características del kernel que hacen posible la virtualización del sistema operativo. En conjunto los Grupos de Control de Linux (de aquí en adelante cgroups) y los Espacios de Nombres (de aquí en adelante namespaces) permiten a un grupo de procesos compartir la plataforma de hardware y el kernel con otros grupos de procesos pero ejecutarse de manera aislada y controlada. Las dos tecnologías seleccionadas como ejemplo de administración de contenedores se apoyan en estas características de kernel.

cgroup permite limitar recursos para un grupo de procesos y namespace permite aislar esos recursos y hace que los procesos los vean como suyos.

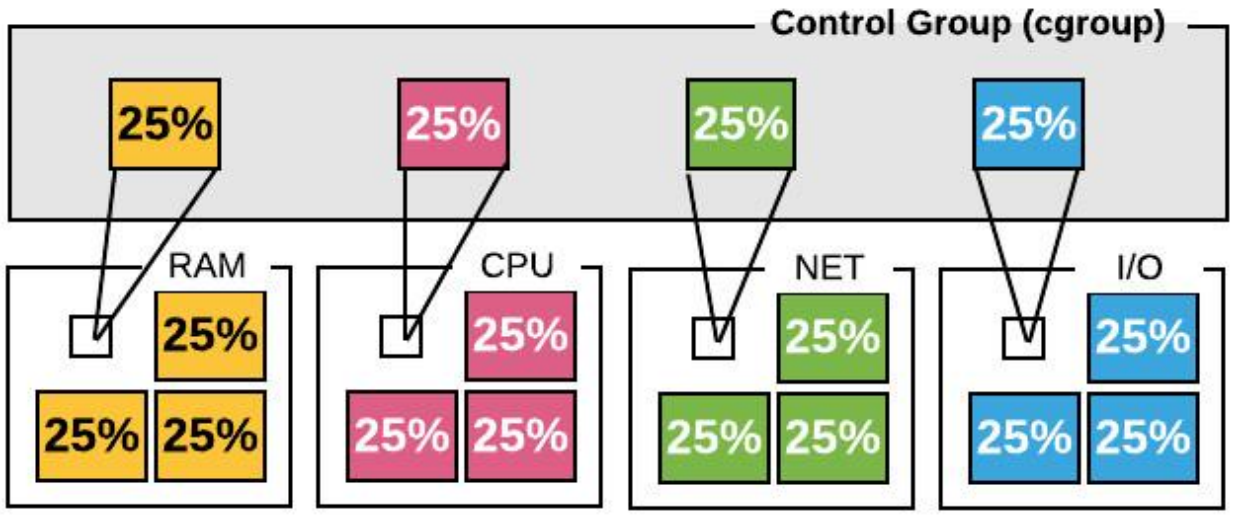




CGROUPS

Grupos de control es la característica de kernel de Linux que limita, aísla y controla el uso de los recursos de un grupo de procesos. Mediante control de

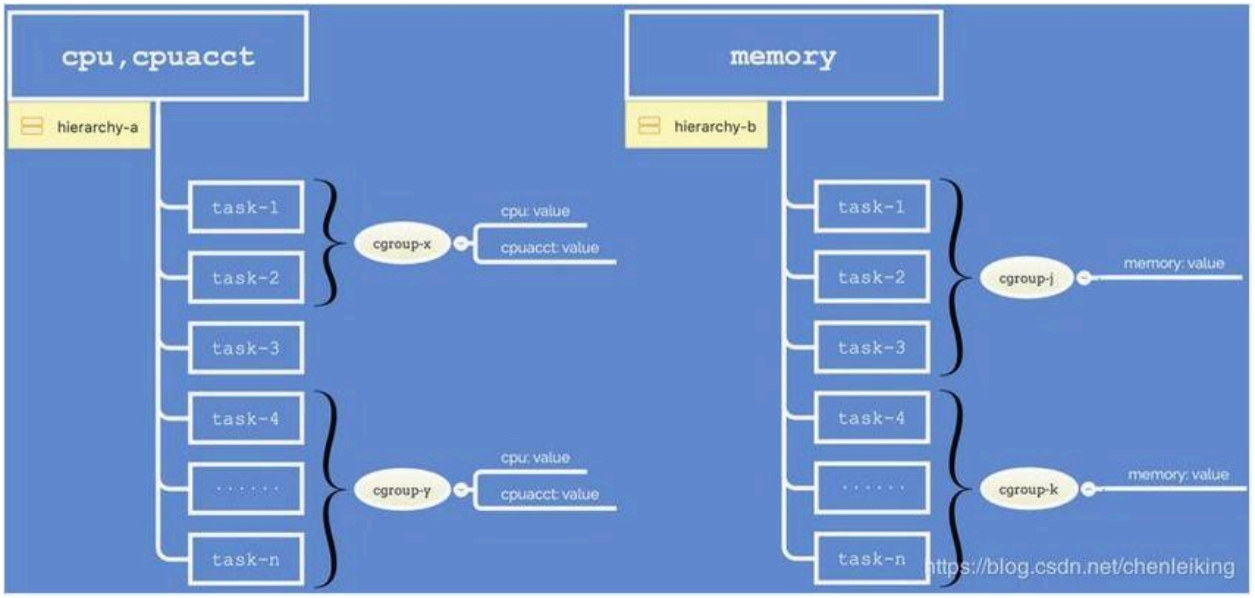
grupos se pueden establecer cuotas de recursos para memoria, CPU, red y entrada/salida. Establecer cuánto de un recursos disponible se le asigna a un proceso o grupo de procesos y además medir dicha cantidad es una parte importante del aislamiento de recursos.



Perspectiva simplificada de control de grupos

Modelo de implementación de CGROUPS

Un cgroup asocia un conjunto de tareas con un conjunto parámetros para uno o más subsistemas. La estructura de un cgroup es jerárquica, donde cada nivel define una serie de parámetros que establecen límites y conteo, referidos al subsistema de tipo de recurso al cual pertenece el cgroup. El esquema de la figura anterior muestra a grandes rasgos los distintos tipos de cgroup que pueden existir. La jerarquía de cgroup es una estructura que permite implementar la lógica de una configuración determinada de límites de recursos en el cgroup raiz del arbol y configuraciones de limites relativas a esta raíz en los cgroups hijos de la raíz.



Modelo de Cgroup

Dicho lo anterior se infiere que habrá una jerarquía por cada tipo de recurso, subsistema o también llamado controlador de recurso. El detalle de esta implementación consiste en un sistema de archivo virtual denominado cgroupfs y es la interface que presenta el kernel a las aplicaciones. Hay un punto de montaje y una jerarquía de directorios específicos que cgroups pone a disposición de las aplicaciones para que estas puedan hacer uso del modelo de agrupado de procesos para límite y control de recursos. La figura siguiente muestra una posible jerarquía de grupos control vista desde la perspectiva del sistema de archivos.

```
[root@localhost ~]# mount --type cgroup
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,xattr,rele
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,memory)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,blkio)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,hugetlb)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,freezer)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,n
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,devices)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,cpuset)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,cpuacc
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,perf_ev
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,pids)
```

Cgroup desde la perspectiva del Filesystem (V1)

Un ejemplo le dará al lector una idea más concreta de cómo el sistema operativo pone a disposición de las aplicaciones la tecnología de grupos de

control. El ejemplo consiste en la creación de un grupo control para el subsistema de memoria y la asociación de un proceso o tarea al mismo. Se examinan los resultados al exceder los límites del grupo de control creado.

Cuando se instala el soporte para cgroups el sistema operativo crea un sistema de archivos virtual y lo monta en el punto /cgroup. A partir de ese momento está disponible para la aplicaciones una estructura jerárquica con los diferentes controladores de subsistemas, montados en puntos de montaje representativos para cada uno de ellos en el primer nivel de la misma, como se vio en las figuras anteriores.

Ahora las operaciones efectuadas contra esta estructura por las aplicaciones tendrán un impacto específico, esta semántica es proporcionada por la implementación de cgroups mientras que la simplicidad está proporcionada por un conjunto de operaciones sobre un sistema de archivos.

Para demostrar los límites impuestos por cgroup en primer lugar se crea una función que incrementa el uso de memoria ram reservando dinámicamente de 1 MB y notificando el resultado de la operación de reserva.

```
#include<iostream>
#include <new>
#include <cstdlib>

int main(){
        int i=0;
        char* ptr =NULL;
        while(i<50){
             if ((ptr =(char*)malloc(1048576)) == NULL) {   ///1MB allocated
               std::cout << "Allocation fails at " << i << "MB\n";
               return 0;
             }

             std::cout << "Allocated "<< i+1 << "MB\n";
             i++;
        }
        std::cout << "Finished allocation";
        return 0;
}
```

Luego se crea un directorio bajo la raíz del subsistema de memoria /cgroup/memory de la siguiente manera:

```
#cd /cgroup/memory

#mkdir prueba
```

Ahora la estructura está formada por un cgroup hijo del subsistema de memoria denominado prueba.

El contenido del cgroup se define creando archivos, dentro del directorio del cgroup, cuyo formato de nombre está preestablecido y debe cumplir con ciertos requisitos. Para el ejemplo, el cgroup de memoria denominado testmem establece un límite de 2 MB en el uso de memoria para todas aquellas tareas que estén asociadas con él. Para ello se crean dos archivos bajo el directorio del cgroup de la siguiente manera:

#echo 2097152 >> /cgroup/memory/testmem/memory.limit_in:bytes

#echo 2097152>>/cgroup/memory/testmem/memory.memsw.limit_in:bytes

Si la aplicación es ejecutada directamente se reservaran los 50MB para finalmente alcanzar la línea que imprime "Finished allocation". La siguiente prueba consiste en poner en ejecución el mismo programa pero esta vez asociado al cgroup testmem:

#cgexec -g memory:testmem <binary_name>

El resultado de esta ejecución es que el proceso reserva memoria dinámicamente hasta alcanzar los 2 MB, cuando intenta sobrepasar este límite desencadena el envío de una señal kill al proceso por parte del kernel, es decir el proceso es terminado.

Tipos de Cgroup

Como se mencionó anteriormente cuando se instala cgroup se crea un punto de montaje para cada subsistema y se monta sobre cada uno de estos el controlador correspondiente para cada subsistema. Los subsistemas son:

- **CPU**
- **CPUACT**
- **CPUSET**
- **MEMORY**
- **DEVICES**
- **FREEZER**
- **NETCLS**
- **BLKIO**
- **PERF_EVENT**

- **NET_PRIO**
- **HUGETLB**
- **PIDS**
- **RDMA**

CGROUP *Versión 2*

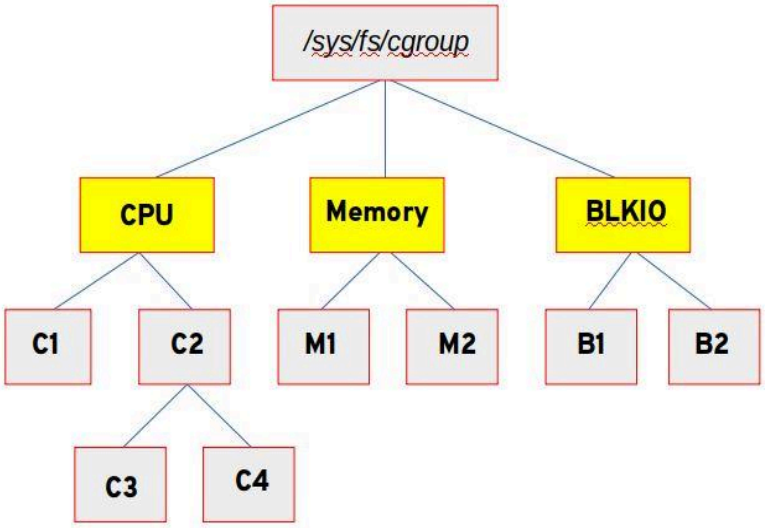
La versión 2 de cgroup presenta algunos cambios respecto de la versión 1 que el autor quiere resaltar.

Múltiples jerarquías versus Jerarquía unificada

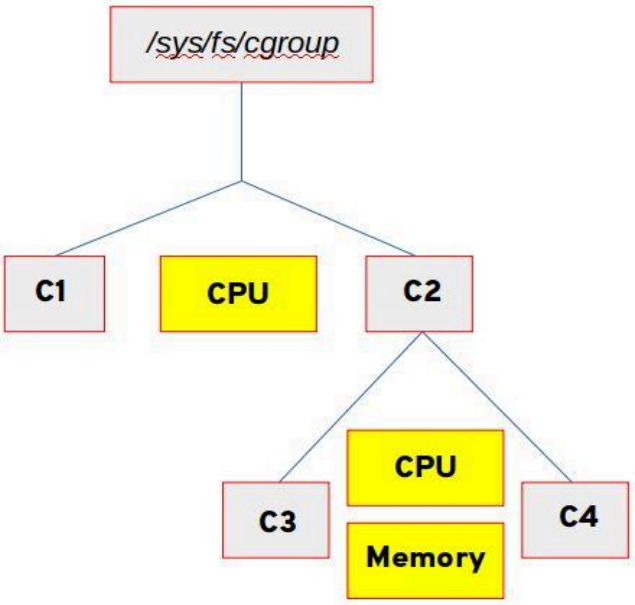
En cgroup v1 cada subsistema tenía su propia jerarquía que el sistema operativo ponía a disposición luego de montar el sistema de archivos virtual de cgroup, bajo una determinada jerarquía que representaba un determinado recursos se creaban cgroups en forma de directorios dentro de los cuales el sistema agrega los archivos necesarios configurar el controlador del recurso en cuestión. Este enfoque daba flexibilidad pero no era muy utilizado y agregaba una excesiva complejidad a nivel kernel.

A partir de la versión 2 de cgroup existe una sola jerarquía, es decir una sola raíz montada en algún punto del sistema de archivos. El contenido de este directorio raíz inicialmente está formado por un conjunto de archivos que controlan el acceso a los recursos por parte del proceso raíz del sistema (al configurar estos archivos se esta configurando los controladores del cgroup raíz, donde estan todos los procesos del sistema). Bajo este directorio se pueden crear subdirectorios que representan sub grupos de control, los cuales al momento de ser creados tienen en su interior los archivos interface asociados a cada subsistema para configurar el controlador correspondiente. Los subgrupos de control son la forma en la que cgroup version 2 proporciona a las aplicaciones un modelo jerárquico de grupos de control, es decir se puede crear una estructura jerárquica de grupos de control. Cada subgrupo de control tendrá todos los controladores de recursos en forma de archivos de interface y un archivo especial llamado cgroup.subtree_control el cual define cuales controladores en el/los grupos de control hijos están habilitados como recursos controlados jerárquicamente, es decir que lo limites que establezcamos en estos controladores serán relativos a su padre siempre que se haya declarado en el padre como un recurso controlado jerárquicamente. Como muestra el ejemplo en la siguiente figura, cgroup versión 2 no tiene un directorio que representa un grupo de control para cada recurso, si no que los directorios representan niveles jerárquicos con archivos de interface en su interior para definir el comportamiento de cada

controlador en el nivel en cuestión y para definir si hereda o no el control de algunos o todos los recursos al subárbol debajo de él.



Cgroup V1



Cgroup V2

El siguiente ejemplo ilustra la creación de un grupo control version 2 con control heredado para el recurso CPU:

En primer lugar se identifican los proceso que van a ser limitados:

```
# top
top - 15:39:52 up  3:45,  1 user,  load average: 0.79, 0.20, 0.07
Tasks: 265 total,   3 running, 262 sleeping,   0 stopped,   0 zombie
%Cpu(s): 74.3 us,  6.1 sy,  0.0 ni, 19.4 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :   1826.8 total,    243.8 free,   1102.1 used,    480.9 buff/cache
MiB Swap:   1536.0 total,   1526.2 free,      9.8 used.    565.6 avail Mem

  PID  USER  PR  NI    VIRT    RES    SHR S  %CPU  %MEM  TIME+ COMMAND
 5473 root    20   0  228440   1740   1456 R   99.7     0.1      0:12.11 sha1sum
 5439 root    20   0  222616   3420   3052 R   60.5     0.2      0:27.08 cpu_load_generator
 2170 jdoe    20   0 3600716 209960 67548 S    0.3    11.2     1:18.50 gnome-shell
 3051 root    20   0  274424   3976   3092 R    0.3     0.2      1:01.25 top
    1 root    20   0  245448  10256   5448 S    0.0     0.5      0:02.52 systemd
...
```

Se monta cgroup versión 2, se definen qué recursos serán controlados jerárquicamente y se crea el grupo de control ejemplo como grupo de control hijo:

```
# mount -t cgroup2 none <MOUNT_POINT>
# ll /cgroups-v2/
-r—r—r--. 1 root root 0 Mar 13 11:57 cgroup.controllers
-rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.max.depth
-rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.max.descendants
-rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.procs
-r—r—r--. 1 root root 0 Mar 13 11:57 cgroup.stat
-rw-r—r--. 1 root root 0 Mar 13 11:58 cgroup.subtree_control
-rw-r—r--. 1 root root 0 Mar 13 11:57 cgroup.threads
-rw-r—r--. 1 root root 0 Mar 13 11:57 cpu.pressure
-r—r—r--. 1 root root 0 Mar 13 11:57 cpuset.cpus.effective
-r—r—r--. 1 root root 0 Mar 13 11:57 cpuset.mems.effective
-rw-r—r--. 1 root root 0 Mar 13 11:57 io.pressure
-rw-r—r--. 1 root root 0 Mar 13 11:57 memory.pressure
# echo "+cpu" > /cgroups-v2/cgroup.subtree_control
# echo "+cpuset" > /cgroups-v2/cgroup.subtree_control
# mkdir /cgroups-v2/Example/
```

Se examina ahora el grupo de control creado:

```
# ll /cgroups-v2/Example/
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.controllers
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.events
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.freeze
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.max.depth
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.max.descendants
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.procs
-r—r—r--. 1 root root 0 Mar 13 14:48 cgroup.stat
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.subtree_control
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.threads
-rw-r—r--. 1 root root 0 Mar 13 14:48 cgroup.type
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.max
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.pressure
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus
-r—r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus.effective
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.cpus.partition
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpuset.mems
-r—r—r--. 1 root root 0 Mar 13 14:48 cpuset.mems.effective
-r—r—r--. 1 root root 0 Mar 13 14:48 cpu.stat
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.weight
-rw-r—r--. 1 root root 0 Mar 13 14:48 cpu.weight.nice
-rw-r—r--. 1 root root 0 Mar 13 14:48 io.pressure
-rw-r—r--. 1 root root 0 Mar 13 14:48 memory.pressure
```

Se completa el ejemplo configurando los archivos de interface del controlador (para cpu_max se especifica cuota y periodo en microsegundos) de CPU en el grupo creado, se asocian dos procesos al subgrupo:

```
# echo "200000 1000000" > /cgroups-v2/Example/cpu.max
# echo "5473" > /cgroups-v2/Example/cgroup.procs
# echo "5439" > /cgroups-v2/Example/cgroup.procs
```

Lo anterior significa que colectivamente dos procesos usarán la CPU durante 0.2 segundos cada 1 segundo. Si se visualiza nuevamente el uso de recursos a través del comando top se nota como los procesos seleccionados han sido limitados:

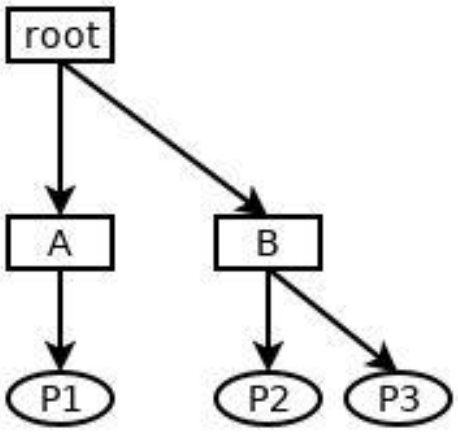
```
# top
top - 15:56:27 up  4:02,  1 user,  load average: 0.03, 0.41, 0.55
Tasks: 265 total,   4 running, 261 sleeping,   0 stopped,   0 zombie
%Cpu(s):  9.6 us,  0.8 sy,  0.0 ni, 89.4 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :   1826.8 total,    243.4 free,   1102.1 used,    481.3 buff/cache
MiB Swap:   1536.0 total,   1526.2 free,      9.8 used.    565.5 avail Mem

  PID USER  PR  NI    VIRT    RES    SHR S  %CPU  %MEM   TIME+ COMMAND
 5439 root  20   0  222616   3420   3052 R  10.0   0.2   6:15.83 cpu_load_generator
 5473 root  20   0  228440   1740   1456 R  10.0   0.1   9:20.65 sha1sum
 2753 jdoe  20   0  743928  35328  20608 S   0.7   1.9   0:20.36 gnome-terminal-
 2170 jdoe  20   0 3599688 208820 67552 S   0.3  11.2   1:33.06 gnome-shell
 5934 root  20   0  274428   5064   4176 R   0.3   0.3   0:00.04 top
 ...
```

Granularidad a nivel de hilo mejorada

Inicialmente únicamente existe el cgroup raiz y a este pertenecen todos los procesos del sistema. Cuando un proceso bifurca un proceso hijo, el nuevo proceso nace en el cgroup al cual pertenece el proceso bifurcador.

Cgroup versión 2 soporta granularidad a nivel de hilo para un subconjunto de controladores para dar soporte a casos de uso que requieren distribución de recursos jerárquica a través de los hilos de un grupo de proceso. Por defecto todos los hilos de un proceso pertenecen al mismo cgroup lo cual también sirve como un dominio de recursos para hospedar el consumo de recursos que no es específico de un proceso o hilo. El modo thread permite a los hilos ser desplegados a través de un subárbol mientras se mantiene el dominio de recursos común para ellos. Este modo es adecuado para controlador de tipo CPU donde tiene sentido aplicar políticas distintas de uso de CPU a diferentes hilos dentro de un mismo proceso.

Marcar un cgroup como cgroup con hilos hace que se una al dominio de recursos de su padre como un cgroup con hilos. El padre puede ser otro cgroup de hilos cuyo dominio de recursos está más arriba en la jerarquía. La raíz del subárbol con hilos, esto es, el ancestro más cercano que no es de tipo con hilos es llamado raíz de hilos y sirve como dominio de recursos para el subárbol entero. El siguiente grafo dirigido es una imagen de una posible distribución de hilos a través de una jerarquía:



Jerarquía de cgroups con procesos en hojas. Dominio de recursos

Namespaces

Otro de los soportes importantes para implementar contenedores son los namespaces. Namespaces permite limitar el ámbito de un recurso global del sistema para un conjunto de procesos y mediante esta abstracción presentar a los procesos su propia instancia aislada del recurso global. Por recurso global se entiende que son todos aquellos recursos de sistemas que son asignados a los procesos y administrados por el kernel. La característica funciona bajo el modelo de un namespace de un tipo determinado el cual se configura con un conjunto de procesos y recursos específicos[1] de ese tipo, donde los recursos específicos referidos en un mismo namespace son distintos. A su vez distintos namespace de un mismo tipo pueden referirse a mismos recursos específicos. Finalmente un proceso no puede estar en más de un namespace de un tipo determinado.

[1] Un recurso específico es una instancia de recurso de un tipo determinado por ejemplo un tipo de namespace posible es MOUNT, el cual puede ser configurado con n de los puntos de montaje que hay en el sistema

Tipos de Namespaces

Los tipos de namespace disponibles en Linux son:

- **CGROUP:virtualiza la vista de los cgroups de un proceso**
- **IPC: aisla ciertos recursos de la comunicación interprocesos**
- **NETWORK: proporciona aislamiento de recursos del sistema relacionados con networking tales como dispositivos de red, pilas de protocolo, tablas de ruteo, reglas de firewall, ect.**
- **MOUNT: proporciona aislamiento de la lista de puntos de montaje vistos por los procesos en cada instancia del namespace.**

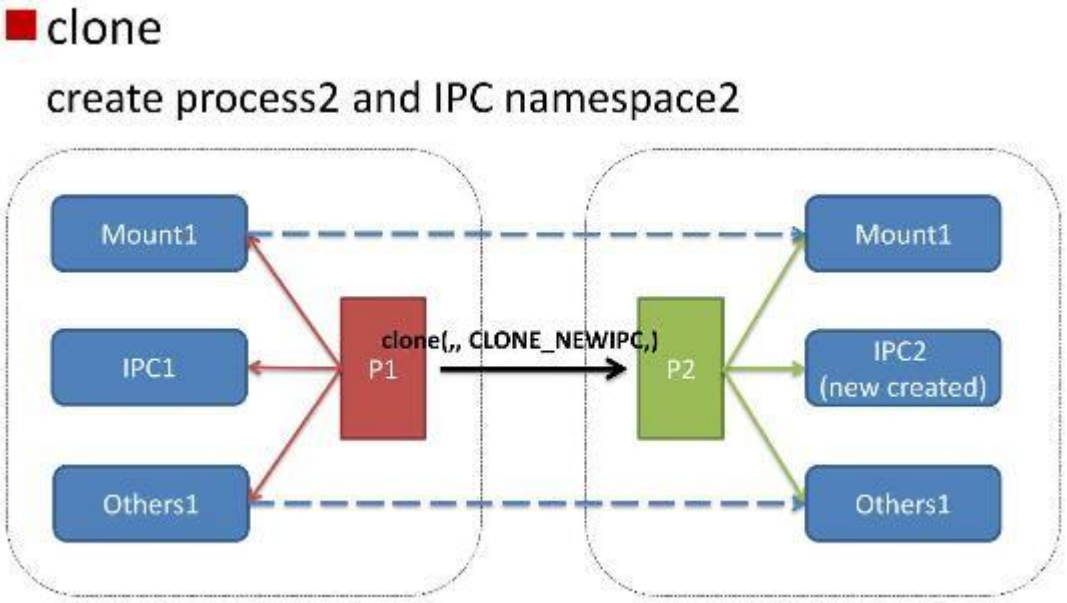
- **PID: aísla el espacio numérico de ID de procesos, lo que significa que procesos en diferentes namespaces PID pueden tener igual PID. Este namespace permite a los contenedores proveer la funcionalidades tales como suspender/continuar el conjunto de procesos del contenedor (debido a la semántica que da el sop al envío de señales por parte de procesos ancestros hacia hijos que estén en un mismo namespace PID) y migrar el contenedor a un nuevo host mientras que los procesos dentro del contenedor mantienen sus PIDs.**
- **UTS:provee aislamiento de dos identificadores de sistemas: el nombre de host y el nombre de dominio NIS.**

La API de namespaces

La API de namespace incluye las siguientes llamadas al sistema, a través de estas queda en evidencia las operaciones fundamentales que se pueden llevar a cabo sobre los namespaces:

- **clone(2): Crea un nuevo proceso hijo (similar a fork() con la diferencia que permite un mayor control sobre cuales piezas del contexto de ejecución son compartidas entre el proceso llamador y el proceso hijo). Si el argumento flags de la llamada especifica uno o más de los flags CLONE_NEW* listados anteriormente, entonces se crean los**

correspondientes nuevos namespaces y el proceso hijo pasa a formar parte de tales namespaces.



Nuevo proceso con su propio recurso IPC, no compartido

- **setns(2):Permite al proceso llamador unirse a un namespace existente. El namespace a unirse se especifica mediante un descriptor de archivo que hace referencia a uno de los archivos *[/proc/[pid]/ns](/proc/[pid]/ns)* que serán explicados a continuación.**

- **unshare(2):Mueve al proceso que llama, a un nuevo namespace. Si el argumento flags de la llamada especifica uno o más de los flags listados anteriormente, entonces el/los nuevo/s namespace/s son creados uno para cada flag especificado y el proceso que llama es unido a tal/tales namespace/s.**

- **ioctl(2): Usado para descubrir información sobre namespaces. Varias operaciones dentro de esta llamada soportan esta funcionalidad.**

Es importante enfatizar que las llamadas clone y unshare son utilizadas tanto para procesos como hilos. Es evidente que clone permite modificar el comportamiento por defecto de que el nuevo proceso hijo forma parte de los mismos namespaces que el padre y permite crear nuevos namespaces siguiendo la jerarquía de procesos (o thread). Por otro lado unshare permite separar selectivamente el contexto de ejecución de un proceso hijo.

Namespace Flags

Los siguientes son los valores posibles en la configuración del parámetro flag presente en las llamadas al sistema de namespace clone y unshare vistas anteriormente:

- **CLONE_NEWCGROUP**
- **CLONE_NEWIPC**
- **CLONE_NEWNET**
- **CLONE_NEWNS**
- **CLONE_NEWPID**
- **CLONE_NEWTIME**
- **CLONE_NEWUSER**
- **CLONE_NEWUTS**

El directorio /proc/[pid]/ns

Para completar la descripción de namespaces, aun cuando no se pretende que esta sea exhaustiva, el autor describe la estructura dentro del sistema de archivos que el kernel presenta a la capa superior para que ésta a su vez presente la abstracción de contenedores.

Los namespaces se implementan como archivos. Además la pertenencia de un proceso a namespaces se implementa mediante enlaces simbólicos asociados a cada proceso que apuntan a cada uno de los namespaces de los cuales forma parte el proceso (la creación de un namespace se corresponde con la apertura de un archivo el cual está disponible para referenciarlo a través de un file descriptor (fd) que puede ser utilizado en una llamada setns). Esta estructura se despliega en el directorio /proc con un subdirectorio para cada proceso cuyo nombre es el identificador del proceso ([pid]) y dentro de cada uno de estos el directorio ns (namespace) el cual contiene un enlace simbólico por cada namespace del que forma parte el proceso. Supongamos un proceso cuyo identificador es 30968, el bloque de la figura muestra el contenido del directorio ns para este proceso.

```
# ls -l /proc/30968/ns/*
lrwxrwxrwx 1 root root 0 Jun  7 14:16 ipc -> ipc:[4026532461]
lrwxrwxrwx 1 root root 0 Jun  7 14:16 mnt -> mnt:[4026532459]
lrwxrwxrwx 1 root root 0 Jun  7 15:41 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 Jun  7 14:16 pid -> pid:[4026532462]
lrwxrwxrwx 1 root root 0 Jun  7 15:41 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jun  7 14:16 uts -> uts:[4026532460]
```

El contenido del link simbólico es una cadena que contiene el tipo de namespace y el número de inodo como lo muestra el siguiente ejemplo, donde mostramos el contenido del link simbólico que representa el namespace uts de la instancia actual de shell:

```
# readlink  /proc/$$/ns/uts
uts:[4026531838]
```

Cgroup Namespace

Es relevante analizar este tipo de namespace debido a que permite abordar la relación entre ambas características. En definitiva, un contenedor es una combinación de namespaces y cgroup para un conjunto de procesos.

Namespace de cgroup virtualiza la vista de los cgroups de un proceso. Los directorios con información de cgroups de un proceso /proc/[pid]/cgroup y /proc/[pid]/mountinfo exponen esta información basándose en este tipo de namespaces.

Cada namespace de cgroup tiene su propio conjunto de directorios raíz de cgroup. Esos directorios raíz son los puntos base para las ubicaciones relativas desplegadas en los correspondientes registros en el archivos /proc/[pid]/cgroup. Cuando un proceso crea un nuevo namespace de cgroup usando clone o unshare con el flag CLONE_NEWCCROUP, sus directorios de cgroup actuales pasan a ser directorios raíz de cgroup del nuevo namespace.

Cuando se leen los miembros cgroup de un proceso objetivo desde /proc/[pid]/cgroup, la ruta mostrada en el tercer campo de cada registro será

relativa al directorio raíz, de la jerarquía de cgroup correspondiente, del proceso lector (es decir relativa al namespace de cgroup del proceso lector). Si el directorio del cgroup del proceso objetivo reside fuera de la raíz indicada por namespace del cgroup del proceso lector entonces el path mostrará entradas ../ para cada nivel superior en la jerarquía de cgroup. The virtualization of /proc/self/cgroup file combined with restricting the view of cgroup hierarchy by namespace-private cgroupfs mount should provide a completely isolated cgroup view inside the container.

El siguiente ejemplo clarifica el concepto:

En primer lugar en el namespace de cgroup inicial se crea un cgroup hijo llamado sub2 dentro de la jerarquía freezer y se ubica dentro de él ,un proceso con PID 20124:

```
# mkdir -p /sys/fs/cgroup/freezer/sub2
# sleep 10000 &     # Crea un proceso que vive por un tiempo
[1] 20124
# echo 20124 > /sys/fs/cgroup/freezer/sub2/cgroup.procs
```

Luego se crea otro cgroup hijo en la jerarquía freezer y se ubica el shell actual dentro de él:

```
# mkdir -p /sys/fs/cgroup/freezer/sub
# echo $$                 # Muestra PID de este shell
30655
# echo 30655 > /sys/fs/cgroup/freezer/sub/cgroup.procs
# cat /proc/self/cgroup | grep freezer
7:freezer:/sub
```

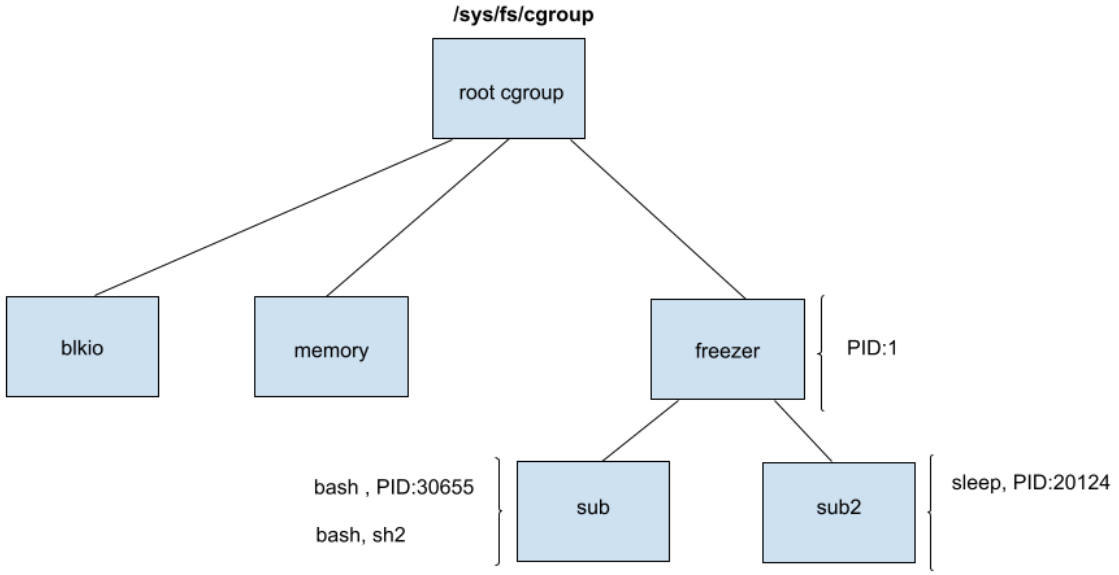
Además se crea un proceso shell mediante unshare y configura los flags de modo que el nuevo proceso sea ubicado en nuevos namespaces de tipo Cgroup y mount:

```
# PS1="sh2# " unshare -Cm bash
```

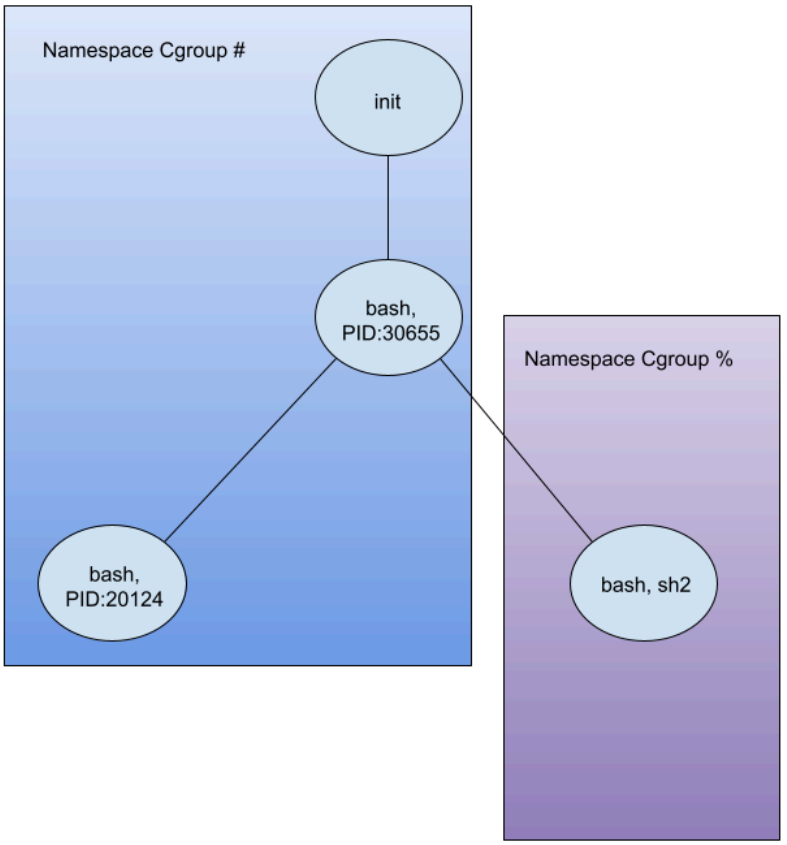
Finalmente, desde el nuevo shell iniciado con unshare, se inspeccionan los archivos /proc/[pid]/cgroup para respectivamente, el nuevo shell, un proceso

que está en el namespace cgroup inicial y el proceso en el cgroup hermano sub2.

```
sh2# cat /proc/self/cgroup | grep freezer
7:freezer:/
sh2# cat /proc/1/cgroup | grep freezer
7:freezer:/..
sh2# cat /proc/20124/cgroup | grep freezer
7:freezer:/../sub2
```



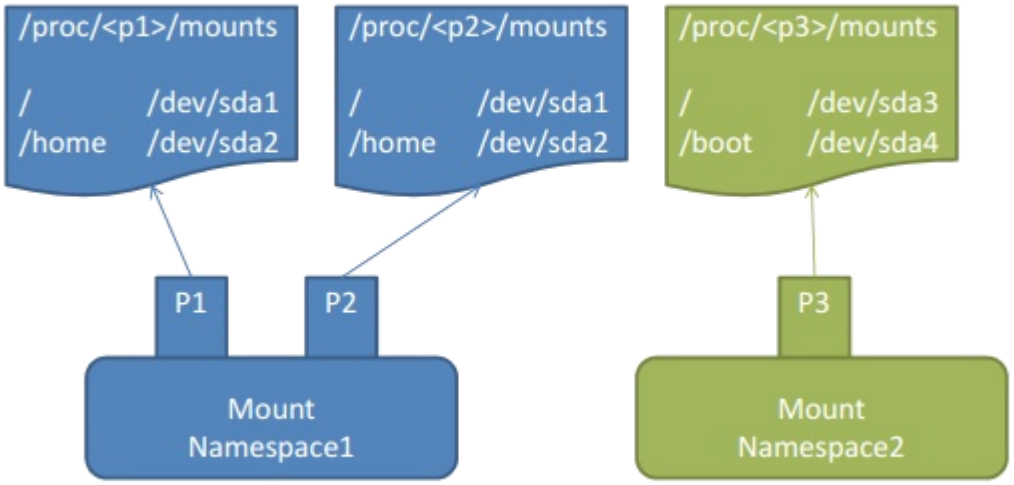
Estructura de cgroups en el ejemplo



Jerarquía de procesos y namespaces en el ejemplo

Namespaces Mount

Los namespaces mount proporcionan aislamiento de la lista de puntos de montaje vistas por los procesos en cada instancia del namespace. Así los procesos en cada una de las instancias del namespace mount verán jerarquías distintas de un solo directorio.



Ejemplo de Namespaces Mount

Es importante aclarar que en las llamadas al sistema para creación de namespaces no se interviene en la configuración del namespaces, la creación de un namespace se da en el contexto de la creación de un proceso y la semántica de acceso a recursos queda definida en términos jerárquicos. Simplemente el llamador indica si quiere crear o no un si quiere crear o no un nuevo namespace o si quiere formar parte de otro namespace. Asimismo la configuración inicial de cada tipo de namespace de un contenedor proviene de lo que declarativamente se especifica, en relación a los recursos que formarán parte de la imagen autocontenida, en el correspondiente archivo de configuración para la construcción de la imagen del contenedor.

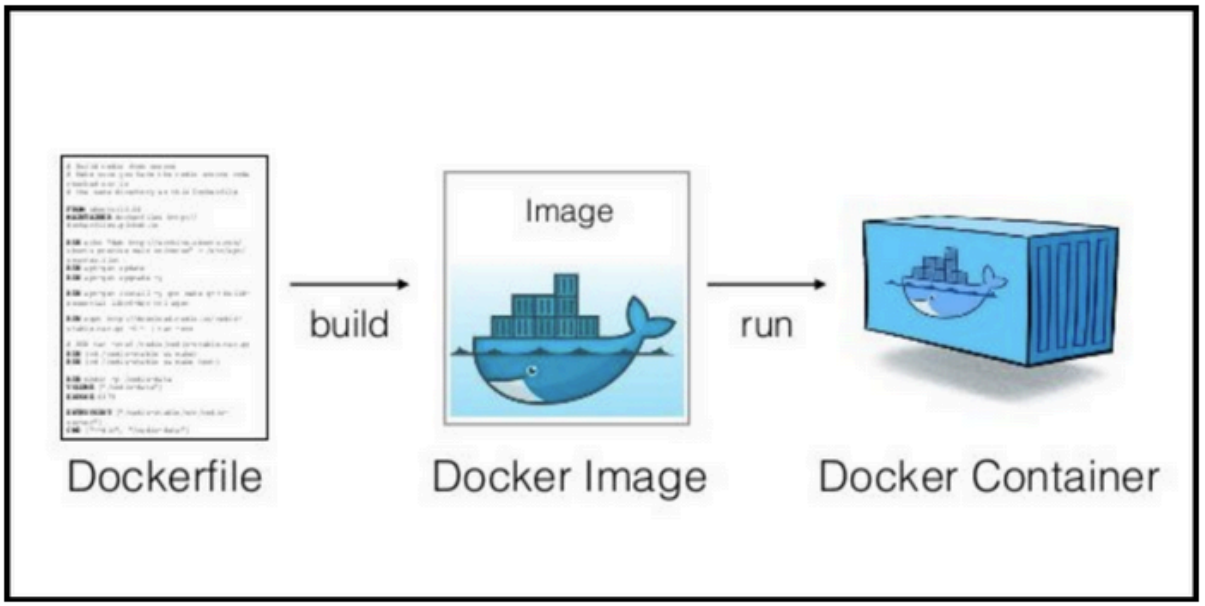
Administración del ciclo de vida de contenedores

Una primer aproximación a tecnología de administración de contenedores: Docker

Como sabemos, microservicio es un concepto arquitectural independiente de la tecnología. No hay una definición estándar clara ni especificación disponible, sin embargo en función de lo que se ha visto en el capítulo anterior definir a los microservicios como unidades de servicio que conforman una aplicación, las cuales son desarrolladas, desplegadas y escaladas independientemente. Varios proveedores ofrecen soporte específico para la construcción de microservicios pero en esencia ellos no están relacionados a tecnología alguna.

Como se mencionó anteriormente, en la actualidad una de las tecnologías de implementación para el modelo de microservicios son los contenedores. Los contenedores de software son unidades de ejecución que llevan empaquetadas en su interior todo lo necesario para que la aplicación containerizada pueda ser ejecutada, esto es, el propio código, librerías del sistema, el entorno de ejecución o cualquier tipo de configuración.

Hablando ya de una tecnología de contenedores específica se toma el caso Docker por ser una de las tecnologías más utilizadas (aclaramos que además de Docker existen otras tecnologías como CoreOs rkt, Mesos, Lxc). En Docker el concepto abstracto de microservicio se implementa, impactando en el desarrollo, despliegue y ejecución (es decir unidades independientes), mediante la creación de la aplicación distribuida en términos de servicios de grano fino y asociación de un archivo descriptor correspondiente a cada microservicio. Docker lanza contenedores basados en imágenes (note que contenedor no es una entidad conocida por el sistema operativo, es docker quien construye esta abstracción para las capas superiores). Una imagen es como un anteproyecto que define lo que debe estar dentro del contenedor una vez que éste sea creado. El modo usual de definir una imagen es a través de Dockerfile. Brevemente, el docker file es una descripción que contiene declaraciones necesarias para adecuar el entorno de ejecución y configurar el servidor donde va a ser escalado el servicio. A partir de este archivo se construye una imagen que contendrá tanto las dependencias como las instrucciones, dirigidas al servidor de contenedores, para que sea éste quien finalmente tome la imagen y la transforme en una unidad en ejecución denominada contenedor. El siguiente gráfico muestra lo anteriormente descripto:



El proceso de creación de una aplicación en contenedor

La construcción de imágenes de contenedores la lleva a cabo DockerDaemon, para construirla ejecuta las instrucciones dirigidas a él en el dockerfile las cuales a su vez hacen referencia a un contexto de construcción, tanto el contexto como el dockerfile son especificados en el comando docker build . Un contexto de construcción es el conjunto de archivos y localizaciones (provenientes de diferentes orígenes) que a través de ciertos parámetros del comando build son proporcionados al demonio Docker. Algunos comandos muy utilizados en un docker file son lo siguientes:

FROM <image> [as <name>]: establece la imagen base para instrucciones subsecuentes, representa un build stage (etapa de construcción). Esta instrucción puede aparecer múltiples veces en un mismo dockerfile para crear múltiples imágenes o para usar un build stage que sea dependencia de otro.

RUN <comando>: ejecuta cualquier comando en una nueva capa encima de la imagen actual y confirma los resultados. La imagen confirmada (que generó un nuevo layer de imagen) resultante será utilizada para el siguiente paso en DockerFile.

COPY <src> <dest>: copia directorios o nuevos archivos y los agrega al filesystem del contenedor en el path <dest>. Se pueden especificar varios recursos <src>, pero las rutas de los archivos y directorios se interpretarán como relativas al contexto de construcción.

CMD <comando> :el propósito fundamental de este comando es definir un comando a ejecutar por defecto, en el momento en que se inicia el contenedor asociado a la imagen en cuestión.

WORKDIR <path>: establece el directorio de trabajo para cualquiera de las instrucciones RUN,CMD,COPY,ADD,ENTRYPOINT.

El siguiente ejemplo consiste en la especificación de un dockerfile, construcción de imagen de contenedor y ejecución de una aplicación nodeJS.

```
FROM node:12-alpine
RUN apk add --no-cache python g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

Ejemplo de Dockerfile

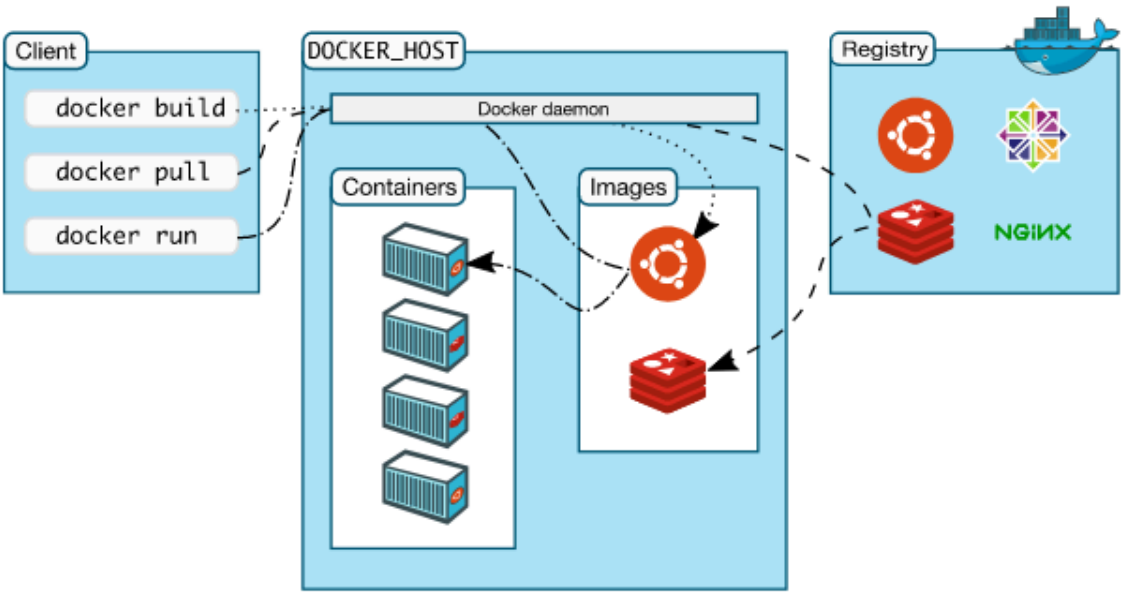
```
docker build -t getting-started

docker run -dp 3000:3000 getting-started
```

Construcción de imagen y ejecución

Arquitectura de Docker

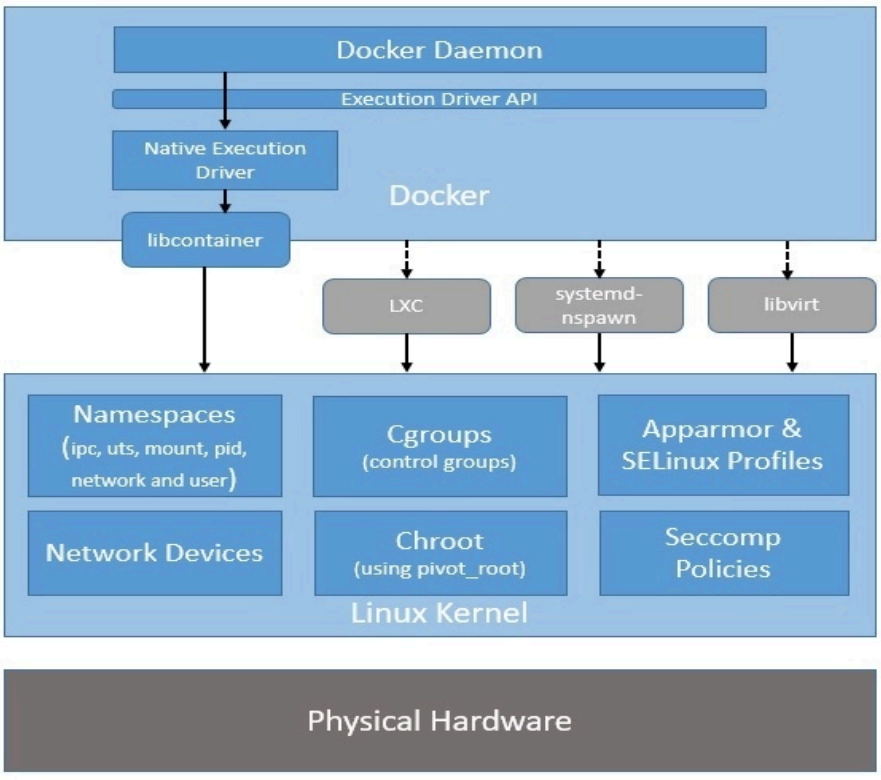
Las imágenes son habitualmente almacenadas en repositorios llamados Docker Registries. DockerHub es un registry de docker público. Para descargar imágenes y poner en ejecución contenedores es necesario tener un DockerHost. DockerHost es una máquina Linux que ejecuta DockerDaemon.

Para lanzar un contenedor se puede usar un DockerClient, quien envía las instrucciones necesarias a DockerDaemon. DockerDaemon también puede interactuar con un Docker Registry en caso de no hallar una imagen solicitada en repositorio local. Docker Registry es un repositorio Docker que funciona bajo el modelo cliente servidor. Una posibilidad es que un usuario ponga en ejecución un Docker registry con sus propias imágenes. La siguiente figura muestra una perspectiva arquitectural de la interacción entre los componentes de Docker.



Interacción entre componentes de Docker

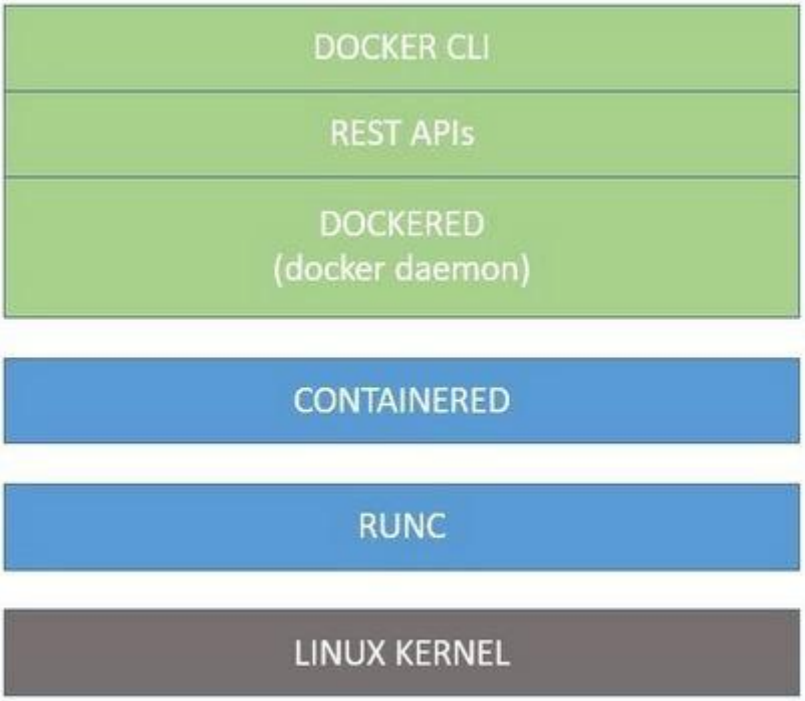
Finalmente se examina la arquitectura de docker desde la perspectiva de la pila de capas que conforman la aplicación, una primer perspectiva es la que muestra la siguiente imagen:



Arquitectura de Docker

Y aquí se ve una de las ventajas de docker, antiguamente docker se apoyaba en LXC y era totalmente dependiente de este. En la actualidad docker es totalmente independiente de LXC (aunque compatible) y gracias al desarrollo de la llamada API de controlador de ejecución sus contenedores pueden ser ejecutados en cualquier plataforma que implemente la API desarrollada por la OCI (Open Container Initiative). En la grafico se muestra a los componentes de docker accediendo a las características de aislamiento del sistema operativo a través del la API y como implementación de esta se muestra a libcontainer. Libcontainer es el entorno de ejecución por defecto provisto por docker para sistemas operativos Linux.

Un perspectiva más detallada de la pila de componentes de docker se muestran en la siguiente gráfica:



Arquitectura de Docker - Detalle

Se sabe que Docker CLI es un componente cliente que ofrece un conjunto de comandos los cuales se traducen en peticiones vía REST API a Docker Server. A continuación se describen componentes de la pila:

- **Dockerd: es el componente central de Docker, también conocido como Docker Engine. Comprende al demonio Docker que escucha las solicitudes que provienen vía API y administra los objetos Docker.**

- **Containerd: para administrar objetos Docker, Dockerd usa Containerd que es otro demonio que ayuda a realizar tareas como descarga de imágenes y ejecutarlas como contenedores. Los más importante es que proporciona una API estándar para que clientes como Dockerd se conecten.**

- **RUNC: finalmente containerd necesita un componente para interactuar con las características de kernel específicas, este componente es conocido como RUNC. RUNC es una implementación de la especificación de runtime de contenedores OCI desarrollada por docker.**

Conclusiones

El modelo arquitectural de microservicios es un modelo que proviene de la ingeniería de software y acompañado por los avances en los kernels de sistemas operativos ha contribuido a la explosion en el uso de contenedores. Sin duda esta combinación en la madurez del paradigma arquitectural optimizado en su implementación por el avance en las tecnologías de kernels y su uso masivo ha empujado al cambio en las prácticas asociadas a la administración ciclo de vida de una aplicación, acelerando los procesos y reduciendo el costo de llevar aplicaciones a entornos productivos. Resulta interesante la reunión de empresas y grupos de desarrollo con peso en área de contenedores interesados en desarrollar avances en la API de acceso a características de núcleo y formato de imágenes teniendo en mente un estándar que permitirá interoperabilidad de contenedores, siendo este otro factor preponderante que fomenta la creación de contenedores sin preocuparse por aspectos relacionados con la plataforma.

Documentación de referencia

Fundamentos de Sistemas Distribuidos - Arquitectura orientada a Servicios
http://www.d-net.research-infrastructures.eu/node/34
Distributed Systems Concepts and Design - Colouris,Dellimore,Kindberg,Blair
Distributed Operating Systems - Robbert Van Renesse y Andrew Tanenbaum
Distributed Systems - Andrew Tanenbaum, Maarten Van Steen - Third Edition

Aislamiento de Recursos - Linux Kernel
https://engineering.squarespace.com/blog/2017/understanding-linux-container-scheduling
https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
https://man7.org/linux/man-pages/man7/namespaces.7.html
https://linuxcontainers.org/
https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html
https://man7.org/linux/man-pages/man7/cgroups.7.html
https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

CASOS de ESTUDIO - Orquestadores
Casos de Estudio - Práctico - Kubernetes
https://kubernetes.io/es/docs/tasks/tools/install-kubectl/

Plataforma de Microservicios - Kubernetes
Conceptos
https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
Docker SWARM
https://www.sumologic.com/glossary/docker-swarm/

Docker
The Docker Book - James Turnbull
Using Docker - Adrian Mouat

Seguridad en Microservicios
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf

Guia de seguridad en contenedores de aplicaciones
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

Mecanismos de seguridad usados en microservicios
(PDF) Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping