Sistemas Operativos
Trabajo Práctico Final
Docker©

Cátedra:
Profesor: Lic. Marcelo Gómez

Ayudantes de cátedra:
Lic. Lucy Marticorena
APU Leandro Luque

Alumnos:
SERRUYA ALOISI, Luciano
TOLEDO MARGALEF, Pablo

5 de julio de 2017

Índice

1. Introducción

Un buen punto de partida antes de empezar a describir la tecnología Docker©, sería hacer una breve introducción a la **virtualización**.

El sitio de RedHat Inc. describe a la virtualización de la siguiente manera:

> La virtualización es una tecnología que permite crear múltiples ambientes simulados o recursos dedicados a partir de un sistema de hardware. Un software llamado *hypervisor* se conecta directamente con ese hardware y brinda la posibilidad de dividir un sistema en varios entornos separados, distintos y seguros conocidos como **Máquinas virtuales**. Dichas máquinas se basan el la habilidad del *hypervisor* para abstraer los recursos del hardware y distribuirlos entre las máquinas virtuales acordemente [3]

Las tecnologías que permiten la virtualización surgieron en la década del 1960, sin embargo obtuvieron mayor popularidad cuando varias empresas tenían que correr software de distintos fabricantes en máquinas de un fabricante en particular (que sólo permitían correr su software).

Utilizando la virtualización, dichas organizaciones podían correr software de distintos fabricantes usando distintos tipos y versiones de sistemas operativos. De esta forma, los servidores se utilizaban más eficientemente, reduciendo costos de compras, de instalación, mantenimiento, y refrigeración.

Los recursos que comunmente se virtualizan son los **servidores**, **sistemas operativos**, y las **redes**.

Virtualizar servidores sirve para maximizar su rendimiento, logrando así poder atender más solicitudes de clientes y poder usar sus componentes para realizar más funciones. Virtualizar sistemas operativos permite tener distintos sistemas operativos corriendo a la vez

(por ejemplo, tener una imagen de Windows, y otra de alguna distribución Linux). La virtualización de redes reduce los componentes físicos necesarios para crear múltiples redes independientes entre sí.

1.1. Máquinas virtuales

Como se describió anteriormente, las máquinas virtuales son una **abstracción de los recursos del hardware**. Son programas que emulan una computadora, dando la sensación de que lo que ejecuta la máquina virtual lo ejecuta directamente sobre el hardware (por ejemplo, la máquina virtual cree que tiene un disco duro de 10GB, mientras que realmente es un archivo más en el filesystem del sistema operativo anfitrión).

La capa intermedia que existe entre la máquina virtual y el hardware real se encarga de crear y administrar los recursos físicos para las distintas instancias de máquinas virtuales que estén corriendo en simultáneo. Esta capa intermedia puede correr encima del sistema operativo anfitrión (como por ejemplo VirtualBox, también llamados *Type 2 Hypervisors*), o se puede ejecutar directamente sobre el hardware (*Type 1 Hypervisors*, por ejemplo KVM)[7]. Esta última opción logra mejores rendimientos que la primera ya que se elimina una capa intermedia (que sería el sistema operativo anfitrión), siendo el *hypervisor* el anfitrión, y las máquinas virtuales corren encima de él.

Esta opción de virtualización consume bastantes recursos de la máquina anfitriona, ya que debe emular el sistema completo, desde los recursos físicos (disco duro, procesador, RAM), hasta el filesystem del sistema operativo huésped.

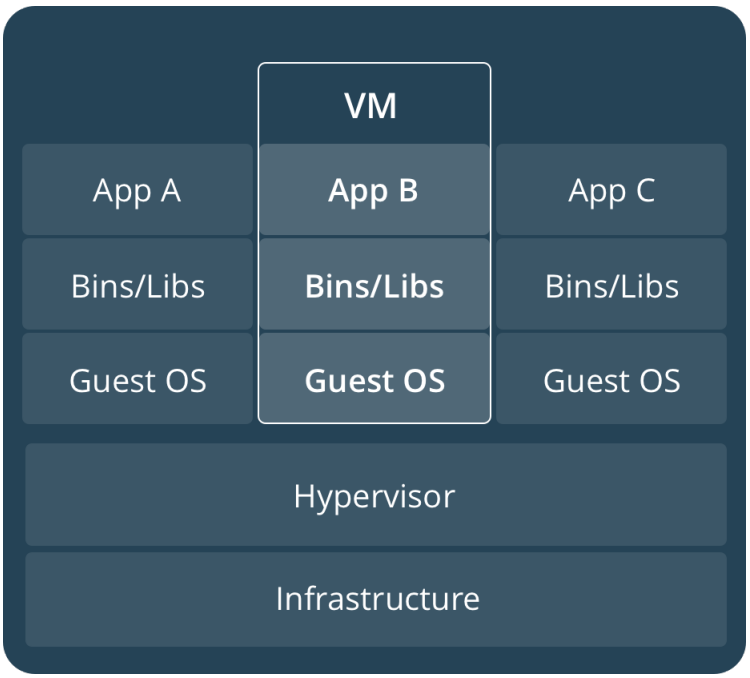

Figura 1: Máquinas virtuales sobre *hypervisor*

1.2. Plataformas de contenedores

Por otro lado existen también los **contenedores**, que el sitio de Docker© los describe como "paquete ejecutable liviano y autónomo de una pieza de software que incluye todo lo que necesita para correr: código, librería del sistema, configuraciones".

Los contenedores son una abstracción en la capa de la aplicación que aislan el software que se desea ejecutar, armando un pequeño ambiente con las dependencias necesarias para que se ejecute solamente ese código. Esto facilita el trabajo de los desarrolladores, sabiendo que al trabajar y probar su producto en un contenedor, ese mismo contenedor podrá ser puesto en producción con la certeza de que funcionará, evitando incertidumbres que pueden generar intentar de ejecutar el software en distintos sistemas operativos o hardware.

El sistema operativo anfitrión sobre el que corren los contenedores restringe el acceso del contenedor a los recursos físicos, por lo tanto un solo contenedor no podrá consumir todos los recursos

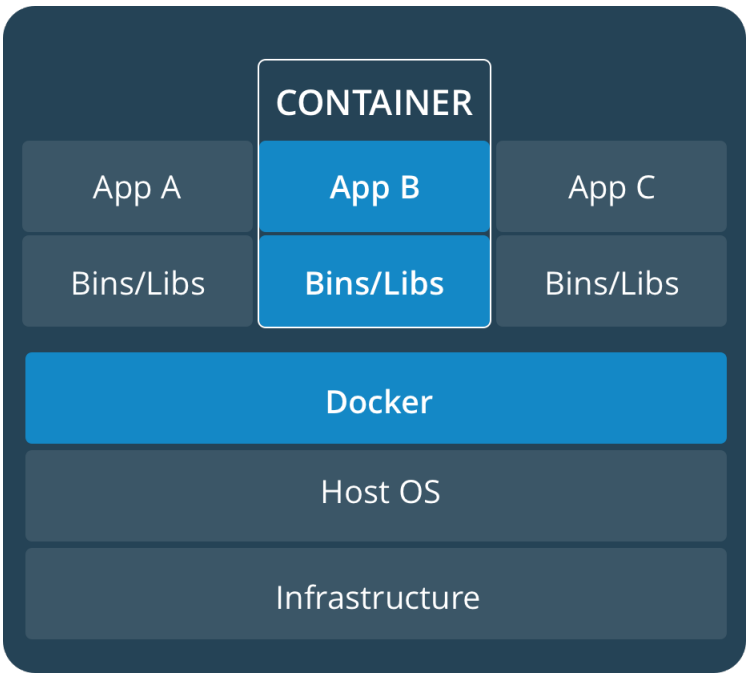

Figura 2: Contenedores corriendo sobre la plataforma Docker©

del anfitrión. Como todos los contenedores comparten el kérnel del sistema operativo anfitrión, logran que su uso (también llamado *virtualización basada en contenedores*) sea más eficiente que el uso de las máquinas virtuales (donde cada máquina virtual ejecutaba un sistema operativo entero)

> La diferencia clave entre los contenedores y las máquinas virtuales es que, mientras el *hypervisor* abstrae un dispositivo entero, los contenedores solo abstraen el kérnel del sistema operativo. [8]

Al momento de momento de decidir si usar una máquina virtual o un contenedor para ejecutar una aplicación de forma aislada, no se debe perder de vista el **alcance** que se espera que tenga la aplicación.[5]

Al ser los contenedores mucho más livianos que las máquinas virtuales (no tienen que correr una instancia del sistema operativo,

sólo la aplicación para la cual fueron diseñados), permiten elevar más el nivel de abstracción, logrando tener un contenedor altamente específico para correr una aplicación en particular. Como práctica general, se mantiene el concepto de tener **un proceso por contenedor**.

Las máquinas virtuales tienen un alcance mucho más amplio, permitiendo correr sistemas operativos enteros.

2. ¿Qué es Docker©?

Docker© es una plataforma de contenedores de código abierto creada en el 2013. La premisa de la empresa es que la herramienta que ellos proveen está destinada a eliminar el problema de "Funciona en mi computadora"[1], al empaquetar el ambiente de ejecución de la aplicación en un contenedor (o imagen) y transportar esa imagen de un servidor a otro.

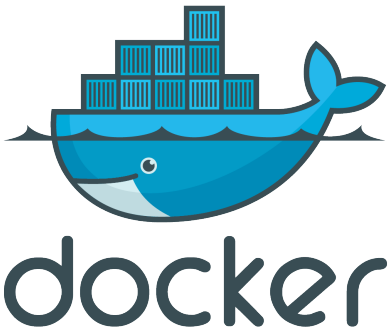

Figura 3: Logo de Docker© - los contenedores que transportan mercadería en barcos representan el mismo concepto de estandarización que los contenedores Docker©

Docker© también permite **apilar** los contenedores, logrando armar el deseado para la aplicación a correr en base a una construcción de distintos contenedores previamente definidos. También permite hacer un gestionado de versiones de las distintas imágenes que se pueden crear, pudiendo así revertir cambios hechos a una imagen.

Docker© es como la *JVM*

> Promesa de Java: *Escribir una vez. Ejecutar en todos lados.*

En vez del código el que debe ser escrito una vez y poder ser

ejecutado en todos lados, las plataformas de contenedores plantean una solución similar pero del lado de la configuración del servidor. Con los contenedores, los desarrolladores pueden tener la confianza que su plantilla de servidor correrá igual en todos los anfitriones donde se ejecute una plataforma de contenedores.[4]

Docker© es como *Git*

> Promesa de Git: *Huellas minúsculas con rendimiento veloz.*

Como se decía anteriormente, Docker© permite hacer un seguimiento de las distintas versiones por las que pasó una imagen (en vez de seguir el código, como hace *Git*).[4]

2.1. ¿Qué NO es Docker©?

A primera vista, Docker© se puede pensar como "máquinas virtuales livianas". Este concepto no es acertado, debido a que, como se explicó anteriormente, las máquinas virtuales difieren de las plataformas de contenedores en su forma de abstraer los recursos y de presentar esa abstracción al usuario.

1. Docker© no es una tecnología de contenedores Linux (como LXC).

2. Docker© no es un reemplazo para un administrador de configuraciones (como pueden ser Chef, Pupper, SaltStack).

3. Docker© no es una *Plataforma como Servicio* (PaaS).[4]

2.2. Arquitectura

La plataforma Docker© emplea los siguientes conceptos para describir su funcionamiento y arquitectura:

1. **Imagen**: definición o plantilla del servidor el cual se virtualizará. Se definen qué dependencias debe tener la imagen.

2. **Contenedor**: instancia de una imagen. Un contenedor corre una imagen en particular.

3. **Docker© Daemon**: procesos que corre en segundo plano y se encarga de administrar los distintos contenedores que se encuentren corriendo en un momento dado.

4. **Dockerfile**: archivo que define la configuración que va a tener una imagen. Emplea una sintaxis específica para su escritura

5. **Docker© Compose**: herramienta que permite definir y correr simultáneamente varios contenedores. También permite crear un archivo de configuración para definir qué imágenes correr.

6. **Local Registry**: repositorio en una red local de la cual los distintos clientes podrán descargar las imágenes que éste publica.

7. **DockerHub**: sitio web que permite subir los repositorios de imágenes y de esta manera se pueden compartir entre distintos usuarios de manera globa.

3. Ventajas y desventajas

Ventajas

Como ya se fue diciendo a lo largo de este trabajo, tanto Docker© como las plataformas de contenedores en general ofrecen varias ventajas sobre las máquinas virtuales u otras técnicas de virtualización. La principal de ellas es que los contenedores son más **livianos** que las máquinas virtuales, permitiendo así poder crear varios a la vez y ejecutarlos en simultáneo. También brinda grandes posibilidades de abstracción de la aplicación a correr, logrando tener distintos contenedores, aislados entre sí, pero que entre todos ofrecen la posibilidad de montar una aplicación.

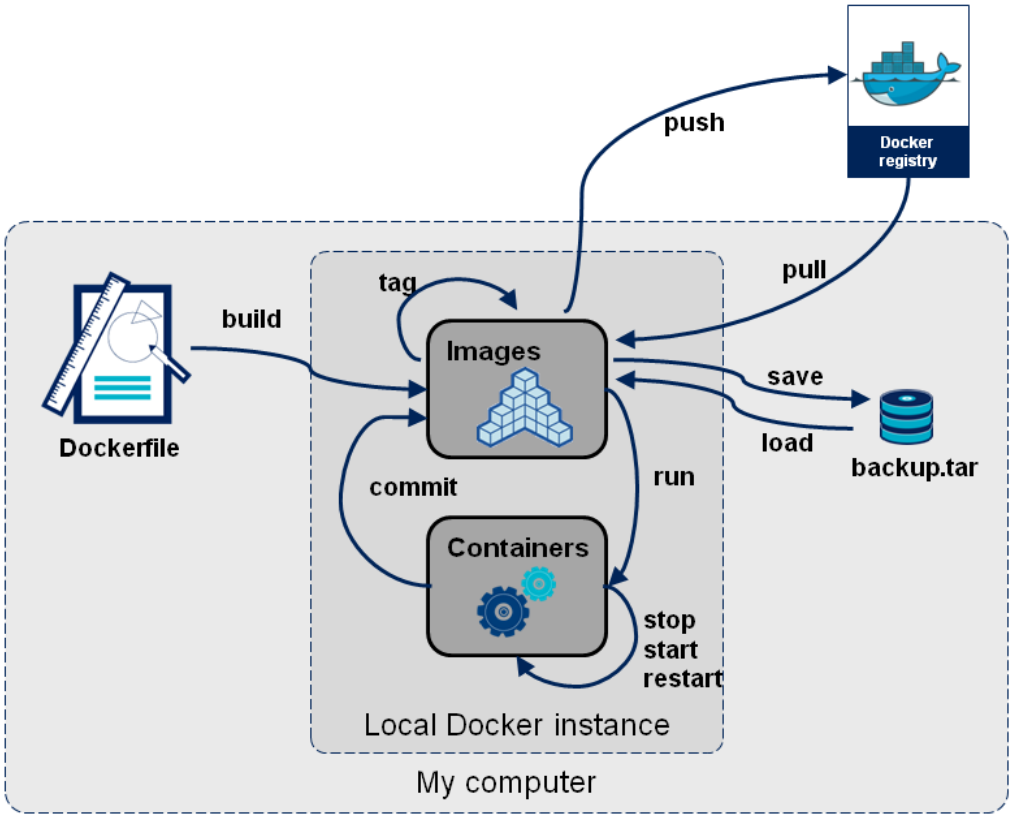

Figura 4: Interacción entre los distintos elementos de la arquitectura de Docker©[6]

La *API* que ofrece Docker© también es muy sencilla, siendo muy intuitivo si ya se cuenta con experiencia en otras herramientas como *Git*

Una de las ventajas más importantes que brinda Docker© que no se debe dejar de lado es la facilidad con la cual los desarrolladores pueden mover su aplicación entre distintos servidores que corran la plataforma, manteniendo así la independencia del hardware.

Desventajas

Más allá de los distintos beneficios y nuevas posibilidades que brindan los contenedores frente a las máquinas virtuales, esta técnica de virtualización también se enfrenta con varias dificultades, siendo uno de los mayores problemas la **incompatibilidad entre distintos sistemas operativos** y la **seguridad**.

Como los contenedores se basan en el sistema operativo anfitrión, utilizan su mismo filesystem para los distintos contenedores que corra la plataforma. Este aspecto es uno de los que brinda las mayores ventajas, ya que es el que evita tener que virtualizar todo el hardware. Sin embargo, imposibilita correr una imagen de un sistema operativo que utilice un filesystem distinto al del sistema operativo anfitrión. Por ejemplo, no se puede correr una imagen de un sistema operativo Linux (ext4) en una máquina con Windows (NTFS). Con respecto a los problemas con la seguridad, vienen dados por una situación: **el demonio de Docker© siempre corre con privilegios de superusuario**. Según la documentación oficial de la herramienta [2], esto sucede porque el proceso demonio se conecta con un socket Unix en vez de con un socket TCP. Por defecto dicho socket Unix es propiedad del usuario *root*. Además, los procesos dentro de los contenedores se ejecutan, por defecto, con permisos de superusuario.

Estas dos características que tiene por defecto Docker© pueden ser problemáticas al momento de que el proceso con permisos de superusuario del contenedor se comunique con el kérnel (que es el mismo kérnel de la máquina anfitrión, porque comparte el sistema operativo).

Una regla básica que se intenta mantener es la de *tratar a los contenedores del mismo modo que se trataría a cualquier servidor de aplicación*. Esto significa:

1. Remover privilegios cuanto antes
2. Correr los servicios del contenedor sin permisos de superusuario siempre que sea posible
3. Tratar al usuario *root* del contenedor como si fuera el usuario *root* de la máquina anfitrión [9]

4. Ejemplo práctico

Para el ejemplo práctico del presente trabajamos decidimos mostrar las bondades de Docker© creando dos imágenes, una con una aplicación en Django© y otra imagen de Postgres©. Luego, con las dos imágenes ya contruidas, pondremos a correr un contenedor de cada imagen.

En este ejemplo utilizaremos también una herramienta que nos proporciona Docker© llamada Docker Compose©, la cual nos permite levantar a la vez más de una imagen en contenedores separados y arma entre estos una red local.

A continuación el Dockerfile correspondiente a la apliación en Django©:

```
# Imagen base a partir de la cual armamos la nuestra
FROM python:3.4-onbuild

# Nos posicionamos en un directorio dentro del contenedor
WORKDIR /usr/src/app

# Copiamos un archivo local hacia dentro del contenedor
# Notar la notacion
# COMANDO <ruta local > < ruta dentro del contenedor >
# Las rutas locales y dentro del contenedor pueden ser tanto
   absolutas
# como relativas
COPY requirements.txt ./

# corremos un comando dentro del contenedor
RUN pip install -r requirements.txt

# Copiamos todo lo que esta en el directorio actual
# al contenedor en su pwd actual
COPY . .

COPY start.sh /start.sh

# Cambia el pwd dentro del contenedor
WORKDIR /usr/src/app/tpfinalso

# El comando entre los [] se ejecutara
# una vez que el contenedor
# se encuentre levantado, a esto se le llama "entry point"
```

```
CMD ["/start.sh"]


EXPOSE 80
```

Como vemos el Dockerfile crea la imagen con todo lo necesario para poder levantar nuestra aplicación. Luego debemos crear la imagen para nuestra Base de Datos, para lo cual creamos la imagen con el siguiente Dockerfile

```
FROM postgres:9.6
```

Ya con las dos imágenes diseñadas podemos crear nuestro archivo de configuración de Docker Compose© llamado docker-compose.yml. Por defecto docker-compose (el comando utilizado para construir aplicaciones de Compose) buscará un docker-compose.yml para realizar la construcción.
En nuestro caso creamos el siguiente docker-compose.yml

```
# version necesaria, nos dice cuales comandos
# y de que forma se deben usar
version: '2'

# Docker Compose le llama a cada contenedor el grupo,
   servicio
services:
    # Servicio con la base de datos
    db:
        # Ubicacion del Dockerfile
        build: ./datos
        # En caso de cierre debe reiniciar el contendor
        restart: always
        # Mapea el puerto 5432 del contenedor al 5434 de
           la maquina local
        ports:
            - "5434:5432"
        # Creacion de volumenes compartidos entre
        # la maquina local y el contenedor
        volumes:
            - ./.pgdata:/var/lib/postgresql/data/

    # Servicio que aloja la aplicacion
    web:
        restart: always
        build: ./server
```

```
ports:
    - "4000:80"
volumes:
    - ./server/tpfinalso:/usr/src/app/tpfinalso
depends_on:
    - db
```

Contando con nuestro docker-compose.yml con la información de los servicios que queremos levantar ejecutamos

```
$ sudo docker-compose build
$ sudo docker-compose up
```

Con esos dos comandos le pedimos al compose que se construya a partir de las info proporcionada en el docker-compose y que comience a correr. En nuestro equipo el build y el up arroja el siguiente resultado en pantalla. Dispondremos sólo de un trozo porque es demasiado para que realmente queremos mostrar:

```
db_1   | LOG:  database system was shut down at 2017-07-04
   13:30:51 UTC
db_1   | LOG:  MultiXact member wraparound protections are now
    enabled
db_1   | LOG:  database system is ready to accept connections
web_1  | Starting Gunicorn
db_1   | LOG:  autovacuum launcher started
web_1  | db.sqlite3
db_1   | LOG:  received smart shutdown request
web_1  | encuesta
db_1   | LOG:  autovacuum launcher shutting down
web_1  | encuesta.backup
db_1   | LOG:  shutting down
web_1  | index.html
db_1   | LOG:  database system is shut down
web_1  | manage.py
db_1   | LOG:  database system was shut down at 2017-07-04
   16:04:17 UTC
db_1   | LOG:  MultiXact member wraparound protections are now
    enabled
db_1   | LOG:  database system is ready to accept connections
```

Como se puede observar el sistema se encuentra corriendo a partir de los dos contenedores y se establece una comunicación entre ellos. Aún así no todo es color de rosa. Al momento de configurar la imagen de la base de datos en importante agregar un volumen entre

la máquina y el contenedor para resguardar la información dentro del contenedor, puede este vuelve al estado que estaba luego de haberse construído por primera vez
Contando con nuestros contenedores corriendo y habiendo configurado correctamente la conexión de la BD con la aplicación tendremos una aplicación aislada del ambiente que contenga los contenedores y la podremos poner a funcionar en otro equipo y no habrá problemas de compatibilidad.

Referencias

[1] . URL https://www.docker.com/.

[2] . URL https://docs.docker.com/engine/installation/linux/linux-postinstall/.

[3] URL https://www.redhat.com/es/topics/virtualization.

[4] Lucas Carlson. What is docker and when to use it, April 2014. URL https://www.ctl.io/developers/blog/post/what-is-docker-and-when-to-use-it/.

[5] Scott Lowe. Virtual machines vs. containers: A matter of scope, May 2014. URL http://www.networkcomputing.com/cloud-infrastructure/virtual-machines-vs-containers-matter-scope/2039932943.

[6] Amaud Mazin. Docker registry first steps, February 2014. URL http://blog.octo.com/en/docker-registry-first-steps/.

[7] Bhanu Tholeti. Learn about hypervisors, system virtualization, and how it works in a cloud environment, September 2011. URL https://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/.

[8] Steven J. Vaughan-Nichols. What is docker and why is it so darn popular?, May 2017. URL http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[9] Daniel J. Walsh. Are docker containers really secure?, July 2014. URL http://blog.octo.com/en/docker-registry-first-steps/.