A Strategy for Container Lifecycle Management

Federico Aguirre, Alfredo Edye, Edgardo Hames
Bitlogic.io

***Abstract*—Virtualization has been around much of the history of computing -from the introduction of virtual memory to virtualization at the operating system level and containers. The use of containers as a deployment tool has boomed since the release of Docker as free software in 2013. Docker includes a large set of tools ranging from building and executing containers on a single node to managing multiple containers in clusters. However, the distribution of deployment descriptors and of maintenance scripts is not properly addressed. This work introduces the mechanisms provided by Docker and describes a practice developed by the Bitlogic team for the deployment and management of the container lifecycle.**

***Index Terms*—Application virtualization, devops, containers, 12-factor apps, Docker**

I. Introducción

Desde comienzos de la década del sesenta, la virtualización ha sido un mecanismo para dividir los recursos de un sistema entre múltiples aplicaciones. La segmentación, paginación, uso de tiempo compartido, virtualización de hardware y sistema operativo, *chroot* (Unix), *jails* (FreeBSD), *zones* (Solaris) y espacios de nombre (Linux) son sólo algunos de los pasos que se han dado en esa dirección [1].

La virtualización a nivel de sistema operativo es un mecanismo de virtualización en el núcleo del sistema operativo que permite la existencia de múltiples instancias de espacio usuario en vez de solo una. Linux ha incorporado diversas funciones (*cgroups*, espacios de nombre, *union-mount filesystem*, etc) que permiten la ejecución de "contenedores" en una única instancia del sistema operativo lo que evita la sobrecarga de ejecutar máquinas virtuales [2].

Google [3] y Facebook [4] fueron dos grandes promotores del uso de contenedores para aprovechar el uso de recursos y simplificar el despliegue de aplicaciones en sus centros de datos.

En el año 2013, Docker aparece en el mundo del software libre como un mecanismo para automatizar el despliegue de aplicaciones dentro de contenedores. El ecosistema de Docker ha crecido hasta incluir herramientas que permiten administrar múltiples contenedores en una sola máquina o la gestión de un cluster con contenedores que corren en distintas máquinas.

En las secciones II, III y IV se describen las herramientas provistas por Docker para la distribución y gestión de contenedores. En la sección V, se presenta un mecanismo para mitigar las limitaciones de Docker en la distribución de los descriptores de entornos y el manejo del ciclo de vida de contenedores.

II. Contenedores

Los contenedores "encierran" un sistema de archivos completo que incluye todo lo necesario para ejecutar un proceso: código, entorno de ejecución, y herramientas y bibliotecas del sistema. De esta manera, los contenedores garantizan que su ejecución será siempre igual independientemente del entorno en el cual se ejecute. Se elimina así el famoso problema "en mi computadora funciona" que atormenta a desarrolladores y testers [5].

Docker implementa una API de alto nivel para contenedores que ejecutan procesos aislados. A diferencia de una máquina virtual, Docker no necesita un sistema operativo independiente. En cambio, utiliza los mecanismos del kernel para aislar recursos (CPU, memoria, red, etc) y separar espacios de nombres (conjuntos de identificadores de procesos, usuarios, nombres de host, etc) [6].

A. Arquitectura de Docker

Docker usa una arquitectura cliente-servidor. El cliente de Docker habla con el servidor de Docker que realiza las tareas pesadas de construir, ejecutar y distribuir los contenedores de Docker. El cliente y el servidor pueden ejecutarse en la misma máquina o el cliente puede conectarse a un servidor remoto. El cliente y el servidor se comunican usando una API REST a través de sockets UNIX o una interfaz de red [7]. La imagen 1 muestra dicha arquitectura.

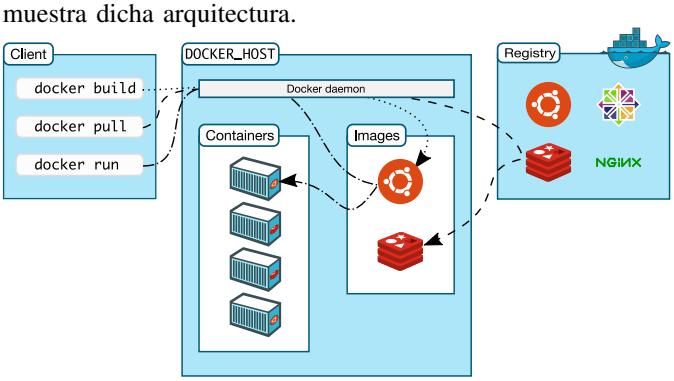

Figure 1: Arquitectura de Docker

B. Conceptos de Docker

Un *contenedor* es una instancia en ejecución de una imagen de Docker, junto con un entorno de ejecución y un conjunto estándar de instrucciones. Una *imagen* de Docker es un colección ordenada de cambios a un sistema de archivos y parámetros para su ejecución en un entorno de contenedores [8].

Una imagen de Docker se construye a partir de las instrucciones provistas en un *Dockerfile* [9]. Un Dockerfile es un

documento de texto que contiene todos los comandos que un usuario ejecutaría para construir una imagen.

```
FROM alpine:latest
COPY svc /opt/bitlogic/search/
ENTRYPOINT /opt/bitlogic/search/svc
EXPOSE 80
```

Listado 1: Ejemplo `Dockerfile`

En el Listado 1, se muestran varios comandos típicos en la construcción de una imagen de Docker:

- FROM especifica la imagen que debe usarse como base.
- COPY indica que se copie un archivo en el path indicado dentro de la imagen.
- ENTRYPOINT declara el comando que debe ejecutarse por default al crear el contenedor.
- EXPOSE indica que el contenedor recibe *requests* en el puerto 80.

El comando `docker build` ejecuta secuencialmente dichas instrucciones para generar una imagen de Docker [10]. Entonces podemos construir una imagen de Docker con el nombre "svc" y versión 1.0:

```
$ docker build -t svc:1.0 .
```

La invocación del comando `docker run` ejecuta el contenedor con la imagen de "svc" versión 1.0:

```
$ docker run --rm svc:1.0
```

Al finalizar la ejecución del *entrypoint*, el contenedor también termina la suya. El sistema de archivos creado para el contenedor quedará disponible tras la ejecución para que pueda examinarse. En algunos casos, no es necesario que los archivos queden disponibles, en cuyo caso pueden eliminarse automáticamente usando la opción `--rm` del comando `run` [11].

III. Múltiples contenedores en un nodo

En la sección anterior, vimos cómo se construye una imagen y se ejecuta un contenedor. En general, los sistemas reales cuentan con más de una aplicación. Por ejemplo, una típica aplicación web incluye al menos un servidor de aplicaciones y una base de datos.

Docker Compose es una herramienta para definir y ejecutar aplicaciones con múltiples contenedores en una computadora [12]. Dichos contenedores se describen en un *archivo de compose* de tipo YAML que define servicios, redes y volúmenes [13].

```
version: '2'
networks:
  dev:
    driver: bridge
services:
  database:
    image: mysql:5.7
    networks:
      - dev
    volumes:
     - dbvol:/var/lib/mysql
  web:
    image: svc:1.0
    networks:
      - dev
    ports:
      - 80:80
    volumes:
      - logvol:/var/log
    depends_on:
      - database
volumes:
  logvol: {}
  dbvol: {}
```

Listado 2: Ejemplo `docker-compose.yml`

En el Listado 2, se describe un archivo `docker-compose.yml`[1] para una aplicación web simple. En éste se define una red virtual denominada *dev* a la cual se conectarán dos servicios: *database* y *web*. También se indica qué imágenes deben usarse para crear los contenedores de los servicios y los volúmenes donde se almacenarán datos. Los volúmenes son esenciales para preservar datos entre distintas ejecuciones de los contenedores (por ejemplo, en el caso de una actualización de los servicios) [14].

Docker Compose incluye diferentes comandos para administrar servicios. También para ver el estado de ejecución de los servicios y sus *logs*.

Por ejemplo, para activar los servicios:

```
$ docker-compose up
```

Para detenerlos:

```
$ docker-compose stop
```

Para consultar el estado de los servicios:

```
$ docker-compose ps
```

[1]Nombre de archivo que Docker Compose usa si no indicamos uno.

IV. Múltiples contenedores en un cluster

En aquellos casos donde las aplicaciones son muy grandes o deben crecer conforme aumente la demanda, los despliegues en un único nodo son insuficientes. Para esos casos, la herramienta Docker Swarm provee mecanismos para la gestión de un cluster de nodos, incremento o reducción de instancias de un servicio, conciliación de estado, descubrimiento de servicios, balanceo de carga y comunicación encriptada entre nodos. La descripción de una aplicación se hace a través de un archivo YAML que extiende la sintaxis de Docker Compose para incluir criterios para replicar los servicios y restricciones sobre los nodos en los cuáles pueden desplegarse.

```yaml
version: '3'
networks:
  dev:
    driver: overlay
services:
  database:
    image: mysql:5.7
    networks:
      - dev
    volumes:
      - dbvol:/var/lib/mysql
    deploy:
      mode: global
  web:
    image: myapp:1.0
    networks:
      - dev
    ports:
      - 80:80
    volumes:
      - logvol:/var/log
    depends_on:
      - database
    deploy:
      placement:
        constraints:
          - node.labels.type == frontend
volumes:
  logvol: {}
  dbvol: {}
```

Listado 3: Ejemplo `swarm.yml`

El Listado 3 extiende el Listado 2 para ser desplegado en un cluster. Docker Swarm crea una red virtual de tipo *overlay* que conecta múltiples nodos. En dicha red, ejecuta la base de datos en modo "global" (una instancia en cada nodo) y la aplicación web en todos aquellos nodos que se hayan configurado como de tipo "frontend".

Para desplegar los contenedores:

```
$ docker stack deploy -c web.yml web
```

Para consultar el estado de los servicios:

```
$ docker service ls
```

V. Bootstrap

En la actualidad, el software generalmente se entrega como servicios llamados aplicaciones web o SaaS (Software as a Service). La metodología de desarrollo "aplicaciones 12-factor" promueve el uso de **formatos declarativos** para automatizar configuraciones, la existencia de un **contrato claro** entre la aplicación y el sistema operativo para mayor portabilidad, la **simplificación del despliegue** en plataformas *cloud*, la **integración continua** para mayor agilidad, y la **escalabilidad** sin cambios significativos en la arquitectura, las herramientas y las prácticas de desarrollo.

Entre las buenas prácticas de la metodología se indica que el código de administración y despliegue debe entregarse junto con el código de la aplicación para evitar inconsistencias fruto de la falta de sincronización. Asimismo, se recomienda que los scripts de administración sean autocontenidos, es decir que no hagan suposiciones respecto a la disponibilidad de bibliotecas o herramientas en el sistema.

Como se puede observar en las secciones anteriores, si bien Docker plantea varias soluciones para el despliegue de contenedores, no ofrece una solución estándar para el despliegue de los scripts de mantenimiento y configuración del entorno.

Una solución a dicho problema es crear un contenedor al cual denominamos *bootstrap* que se encargue de

1) la distribución de scripts de mantenimiento y archivos YAML para Docker (Compose o Swarm);
2) la descarga de las imágenes del sistema;
3) la gestión de su ciclo de vida (inicio, detención, consulta de estado), y
4) las tareas de mantenimiento (*upgrade*, *downgrade*, *rollback*, etc).

A continuación, se muestran posibles invocaciones del contenedor bootstrap para diversos escenarios. Por ejemplo, para descargar las imágenes del sistema:

```
$ docker run --rm project/bootstrap pull
```

Para desplegar los contenedores del sistema:

```
$ docker run --rm project/bootstrap up
```

Para detener todos los contenedores del sistema:

```
$ docker run --rm project/bootstrap stop
```

Una de las funciones de Docker Compose es detectar cuando la imagen correspondiente a un contenedor cambió y recrearlo. Gracias a esto, es sencillo hacer una actualización de los servicios usando los mismos comandos mostrados anteriormente:

```
$ docker run --rm project/bootstrap pull
$ docker run --rm project/bootstrap up
```

El uso de la opción `--rm` elimina el container de bootstrap tras la ejecución. Para simplificar aún más la invocación, podemos definir una función de Bash:

```
$ bootstrap() { docker run --rm project/
    bootstrap $1 }
```

Los comandos entonces son más sencillos de recordar y de usar:

```
$ bootstrap pull
$ bootstrap up
```

Como podemos observar, los comandos utilizados para gestionar los servicios proveen una abstracción respecto del mecanismo de despliegue subyacente.

A. Implementación

En esta sección, se muestra una posible implementación del contenedor bootstrap a través de un sencillo script de Bash. En algunos casos, será necesario recurrir a otros lenguajes de programación para mayor flexibilidad.

```
#!/bin/bash
composefile=/opt/bitlogic/compose.yml
command=$1
case ''$command'' in
  ''pull'')
     docker-compose pull
     ;;
  ''up'')
     docker-compose up
     ;;
  ''stop'')
     # Pedir al usuario que confirme
     docker-compose stop
     ;;
esac
```

Listado 4: Script `bootstrap.sh`

Y su Dockerfile correspondiente:

```
FROM docker/compose:1.12.0
COPY compose.yml /opt/bitlogic
COPY startup.sh /opt/bitlogic
RUN chmod 755 /opt/bitlogic/startup.sh
ENTRYPOINT /opt/bitlogic/startup.sh
```

Listado 5: Dockerfile `bootstrap`

Esta implementación de ejemplo tiene solo los comandos básicos mostrados en la sección anterior. No obstante, otros comandos que resultan útiles de implementar son:

- *ps* que consulta el estado de las aplicaciones;
- *logs* que muestra los logs de la aplicación;
- *upgrade* que combina `pull` y `up`;
- *migrate* que actualiza el modelo de datos;
- *rollback* que despliega una versión previa del software, y
- *scale* que modifica el número de instancias de una aplicación.

Una propiedad de este mecanismo es que la interfaz de operación no cambia entre distintos modos de despliegue; los comandos no dependen de la cantidad de nodos. El Dockerfile que se usó para este ejemplo está basado en una imagen que ya incluye Docker Compose. Esto podría modificarse fácilmente en caso de usar Docker Swarm.

B. Ventajas

En esta sección, se enumeran algunas de las ventajes de usar el container *bootstrap*. La primera y más evidente es que el método no está acoplado a un proyecto en particular. Por el contrario, puede aplicarse a cualquier tipo de proyecto que use Docker como tecnología de despliegue.

Al usar Docker, permite hacer despliegues remotos declarando el nodo destino con la variable `DOCKER_HOST`. Siempre con una interfaz de comandos uniforme.

El mecanismo de despliegue está bajo control de configuración. De este modo, se puede garantizar la sincronización entre el código productivo y el de administración. También se puede recuperar la "receta" de despliegue de versiones previas del producto.

Por último, al estar completamente automatizado, es muy fácil de usar con herramientas de integración y despliegue continuo (por ejemplo, Jenkins, Travis, Drone.io, etc). No es necesario duplicar esfuerzo y se minimizan las diferencias entre el entorno de desarrollo y el de producción.

C. Comparación con otras herramientas

En la actualidad, el mercado ofrece muchas herramientas que permiten administrar equipos y orquestar despliegues. Tal es el caso de Puppet [15], Chef [18], Salt [16] y Ansible [17], solo por citar algunos.

Todas estas herramientas son muy potentes, de propósito general y han sido creadas para administrar una gran cantidad de servidores. Algunas requieren instalación de agentes adicionales en cada servidor. Por el contrario, bootstrap es de propósito muy específico y solo requiere la instalación de Docker, el cual estaría presente al querer ejecutar contenedores.

Por otro lado, cada una de ellas requiere el aprendizaje de un nueva tecnología (comandos, archivos de configuración, etc). En nuestra experiencia, bootstrap es desarrollado por el mismo equipo del producto usando las mismas herramientas que ya conoce (lenguaje de programación, librerías de terceros, etc).

VI. Conclusión

Al demandar un menor uso de recursos, los contenedores parecen ser el nuevo reemplazo de las máquinas virtuales para consolidar infraestructura. Docker ha facilitado el acceso a

contenedores de la comunidad de desarrollo de software en general. El ecosistema ha crecido a tal punto que simplifica el manejo de aplicaciones pequeñas o con gran cantidad de componentes.

Sin embargo, el problema de la distribución no parece estar completamente resuelto a pesar de haber buenas prácticas establecidas en la industria. Bootstrap se creó como una solución que integra dichas consideraciones.

Otro aspecto interesante de Bootstrap es que baja la barrera de entrada al uso de contenedores y reduce la curva de aprendizaje de Docker.

Agradecimiento

Agradecemos al equipo de Bitlogic que ha contribuido con ideas y con la revisión del documento.

References

[1] https://en.wikipedia.org/wiki/Timeline_of_virtualization_development
[2] https://en.wikipedia.org/wiki/Operating-system-level_virtualization
[3] https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/
[4] https://blog.docker.com/2014/07/dockercon-video-containerized-deployment-at-facebook/
[5] https://blog.codinghorror.com/the-works-on-my-machine-certification-program/
[6] https://www.docker.com/what-docker
[7] https://docs.docker.com/engine/docker-overview/#docker-architecture
[8] https://docs.docker.com/engine/reference/glossary
[9] https://docs.docker.com/engine/reference/builder/
[10] https://docs.docker.com/engine/reference/commandline/build/
[11] https://docs.docker.com/engine/reference/run/
[12] https://docs.docker.com/compose/
[13] https://docs.docker.com/compose/compose-file/
[14] https://docs.docker.com/compose/overview/#preserve-volume-data-when-containers-are-created
[15] https://puppet.com
[16] http://saltstack.com
[17] https://www.ansible.com
[18] https://www.chef.io