A Strategy for Container Lifecycle Management

Federico Aguirre, Alfredo Edye, and Edgardo Hames

Bitlogic.io, Córdoba, X5009ABY, Argentina,
`info@bitlogic.io`,
Website: `https://bitlogic.io`

Resumen 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.

Keywords: Application virtualization, devops, containers, 12-factor apps, Docker

1. 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 recursos y simplificar el despliegue de aplicaciones en sus centros de datos. Google publicó dos trabajos en los que se describen sistemas de gestión de contenedores: uno enfocado en Omega, un planificador flexible y escalable para grandes *clusters* [5]; el otro, Borg, su herramienta de gestión de grandes *cluster* [6]. Entre ambas publicaciones, liberó el código de Kubernetes, una herramienta de gestión para *clusters* más modestos [7]. Por su parte, en 2014,

Facebook presentó Tupperware, su herramienta para gestión de contenedores y *sandboxes*.

En el año 2013, Docker aparece en el mundo del software libre como un mecanismo para automatizar el despliegue de aplicaciones dentro de contenedores. Provee una capa de abstracción y automatización de los mecanismos de virtualización a nivel de sistema operativo, tanto en Linux como Windows. 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 2, 3 y 4 se describen las herramientas provistas por Docker para la distribución y gestión de contenedores. En la sección 5, 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.

2. 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 [8].

Múltiples contenedores pueden ejecutarse en una misma computadora y compartir un mismo sistema operativo con otros contenedores. Cada uno de ellos funciona como un proceso separado en espacio de usuario. En general, los contenedores ocupan menos espacio (decenas de megabytes) que una máquina virtual e inician casi instantáneamente. Al no existir una capa de virtualización de *hardware*, la sobrecarga de los contenedores es mínima. La figura 1 compara las capas de software en un sistema que utiliza máquinas virtuales y otro con contenedores.

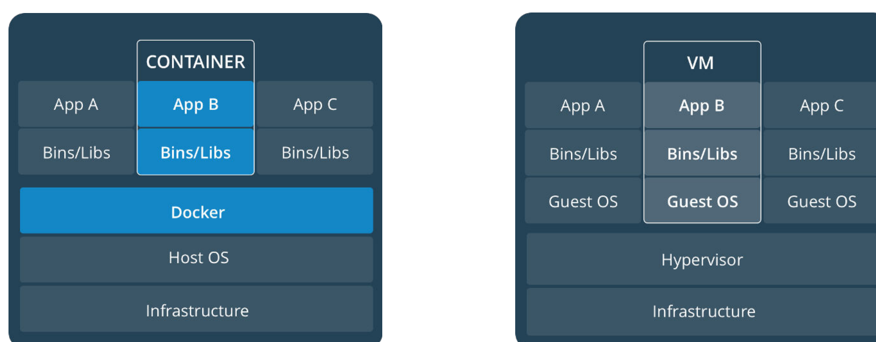

Figura 1: Comparación entre contenedores (izq.) y máquinas virtuales (der.)

Docker implementa una API de alto nivel para contenedores que ejecutan procesos aislados, y 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) [9].

2.1. 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 [13].

Una imagen de Docker se construye a partir de las instrucciones provistas en un *Dockerfile* [14]. Un Dockerfile es un documento de texto que contiene todos los comandos que un usuario ejecutaría para construir una imagen.

Las imágenes de Docker pueden crearse a partir de otra imagen existente a la cual denominamos *imagen base*. Por ejemplo, se puede crear una nueva imagen configurando y personalizando otra ya disponible.

Los contenedores son efímeros y los datos no son persistentes. Para resolver este problema, Docker introduce la idea de *volumen* [10]. Los volúmenes son administrados completamente por Docker y permiten almacenar y compartir datos en la computadora local o en servicios remotos (por ejemplo, NFS, Amazon S3).

Docker también permite crear redes definidas por software y asignar puertos de contenedores a los de la computadora donde se ejecutan. Estas redes pueden extenderse a varias computadoras y cuentan con su propio mecanismo de ruteo y DNS [11].

2.2. Arquitectura de Docker

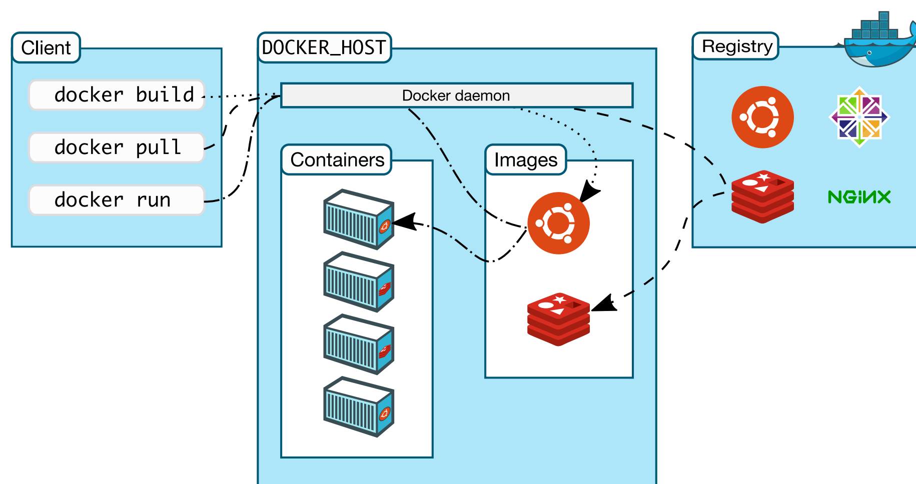

Figura 2: Arquitectura de Docker

Docker usa una arquitectura cliente-servidor. El cliente de Docker envía comandos al servidor de Docker para que éste realice las tareas pesadas de construir, ejecutar y distribuir imágenes. 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 [12]. Un registro (*registry*) de Docker almacena y envía imágenes a los clientes. En la figura 2 se muestra dicha arquitectura.

2.3. Ejemplo

En esta sección, presentamos un ejemplo de una aplicación web que se extenderá en secciones siguientes. Asumiremos que el servicio se compila en único ejecutable (`svc`) que puede correr en Linux. El listado 1.1 muestra un Dockerfile sencillo que sirve para empaquetar y distribuir el backend de dicha aplicación.

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

Listado 1.1: Ejemplo `Dockerfile`

En el Listado 1.1, observamos varios comandos típicos en la construcción de una imagen de Docker:

- `FROM` especifica la imagen base a partir de la cual se construye.
- `COPY` copia un archivo o directorio desde el espacio de trabajo hacia la ruta especificado dentro de la imagen.
- `ENTRYPOINT` especifica el comando que debe ejecutarse por defecto al crear el contenedor.
- `EXPOSE` indica que el contenedor recibe *requests* en el puerto 80 (TCP por defecto).

El comando `docker build` ejecuta secuencialmente instrucciones las instrucciones de un Dockerfile para generar una imagen de Docker [15]. Entonces podemos construir una imagen de Docker con el nombre “svc” y etiqueta 1.0 con los contenidos del directorio actual con el siguiente comando:

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

Podemos ver la lista de imágenes que están descargadas en el demonio de Docker con el comando `docker images`.

```
$ docker images
REPOSITORY      TAG     IMAGE ID        CREATED          SIZE
svc             1.0     d5f8a690b328    5 seconds ago    20MB
```

La invocación del comando `docker run` ejecuta el contenedor con la imagen que creamos en los pasos anteriores:

```
$ docker run -p 8080:80 --rm svc:1.0
```

En este ejemplo, le solicitamos a Docker que redirija los *requests* del puerto 8080 de la computadora al puerto 80 del contenedor. Nótese que así podemos mapear múltiples servicios que usan el puerto 80 en un contenedor dentro de la misma computadora, cada uno de ellos con un puerto distinto.

Al finalizar la ejecución del *entrypoint*, el contenedor también termina de ejecutarse. 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` [16].

También podemos verificar el estado del contenedor usando `docker ps`:

```
$ docker ps --format "table {{.ID}}\t{{.Status}}\t{{.Image}}\
   t{{.Ports}}"
$ docker ps
CONTAINER ID   STATUS         IMAGE     PORTS
e85c9e480ff8   Up 2 minutes   svc:1.0   0.0.0.0:8080->80/tcp
```

En la captura anterior, se muestra un ejemplo de salida formateada para que muestre información reducida sobre el estado del contenedor. En otros casos, la invocación del comando `docker ps` sin parámetros puede ser más útil.

3. 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 un servicio. Por ejemplo, la aplicación web de ejemplo probablemente incluye una base de datos. En estos casos, no es práctico ejecutar manualmente los comandos para iniciar todos los servicios. Además, si debemos desplegar un sistema en varios entornos, quizás necesitemos configuraciones diferentes para cada uno de ellos.

Docker Compose es una herramienta para definir y ejecutar aplicaciones con múltiples contenedores en una computadora [17]. Dichos contenedores se describen en un *archivo de compose* de tipo YAML que define servicios, redes y volúmenes [18]. A su vez, esto nos permite mantener distintos archivos de *compose* para cada uno de los entornos donde debamos desplegar nuestro sistema.

En el Listado 1.2, 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

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

entre distintas ejecuciones de los contenedores (por ejemplo, en el caso de una actualización de los servicios) [19].

```
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:
      - 8080:80
    volumes:
      - logvol:/var/log
    depends_on:
      - database
volumes:
  logvol: {}
  dbvol: {}
```

Listado 1.2: Ejemplo `docker-compose.yml`

Docker Compose incluye diferentes comandos para administrar y verificar servicios. Por ejemplo, para iniciar los servicios:

```
$ docker-compose up
Creating network "svc_dev" with driver "bridge"
Creating volume "svc_logvol" with default driver
Creating volume "svc_dbvol" with default driver
Creating svc_database_1 ...
Creating svc_database_1 ... done
Creating svc_web_1 ...
Creating svc_web_1 ... done
```

Para detenerlos:

```
$ docker-compose stop
Stopping svc_web_1 ... done
Stopping svc_database_1 ... done
```

Para consultar el estado de los servicios:

```
$ docker-compose ps
   Name              Command        State         Ports
-------------------------------------------------------------
svc_web_1       /opt/bitlogic/...    Up      0.0.0.0:80->80/tcp
svc_database_1 /usr/local/bin...    Up      80/tcp
```

4. 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. En algunos casos, querremos que haya una cantidad fija de servicios; en otros, que haya tantos como nodos físicos requiera la solución. 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.

4.1. Arquitectura de Swarm

Un *swarm* (*cardumen*) consiste de varios nodos con Docker que se ejecutan en *modo swarm* y actúan como *managers* o *workers*. Un nodo puede tener uno o ambos roles simultaneamente. Cuando se crea un servicio, el operador define el número óptimo de réplicas, redes, volúmenes y puertos. Docker mantiene dicho estado automáticamente. Así, si un nodo *worker* no está disponible (por ejemplo, está apagado), entonces Docker reasigna sus tareas a otro nodo. En la figura 3, se muestra un diagrama de dicha arquitectura.

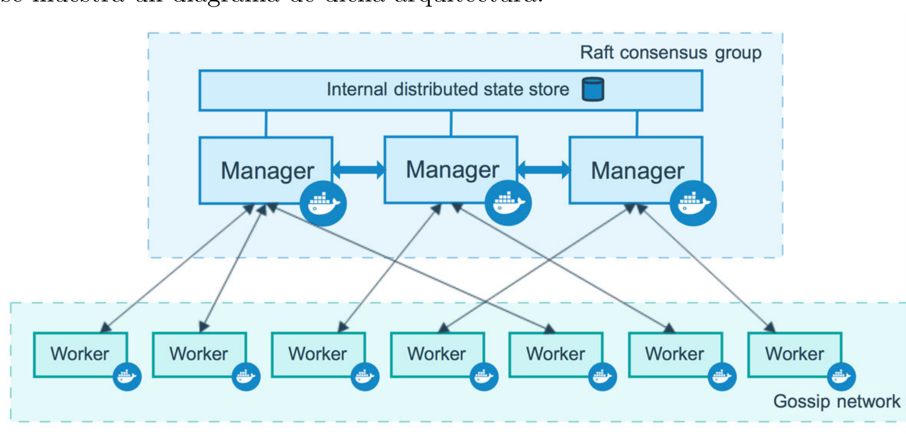

Figura 3: Arquitectura de Docker Swarm

4.2. Ejemplo

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.

```
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 1.3: Ejemplo `swarm.yml`

El Listado 1.3 extiende el Listado 1.2 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". En nuestro ejemplo, usamos el siguiente comando para desplegar los contenedores en un swarm:

```
$ docker stack deploy -c swarm web
```

5. 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. Dicho de otra manera, si bien ofrece mecanismos para la gestión de las imágenes y contenedores, no incluye algo equivalente para la distribución de los archivos YAML de Docker Compose o Swarm. Tampoco para scripts de mantenimiento de bases de datos.

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 y Swarm 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.

5.1. 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'')
     # Opcional: pedir al usuario que confirme
     docker-compose stop
     ;;
esac
```

Listado 1.4: Script `bootstrap.sh`

Y su Dockerfile correspondiente:

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

Listado 1.5: 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.

Si bien la metodología de "12 factores" exige reducir la disparidad entre los entornos de prueba y los de producción, esto no siempre es posible. En ese caso, cuando hay configuraciones que son diferentes según el entorno, las almacenamos dentro de la misma imagen de *bootstrap*. También agregamos un parámetro que nos permita seleccionar el entorno al momento de instanciar el contenedor. Por ejemplo:

```
$ bootstrap up testing
```

5.2. Bootstrap y Kubernetes

En aquellos proyectos donde los servicios se despliegan usando Kubernetes, también hemos utilizado la idea de un contenedor *bootstrap* como interfaz para:

- crear, configurar y eliminar los nodos del cluster;
- instalar y desinstalar la aplicación;
- migrar bases de datos,
- y dar de alta o eliminar usuarios del sistema.

Siguiendo esta misma línea, si el sistema debe desplegarse en distintas plataformas (por ejemplo, porque cada cliente tiene requerimientos diferentes), entonces la estrategia es crear un contenedor de *bootstrap* para cada una de ellas y todos con la misma interfaz de comandos.

5.3. 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.

5.4. 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 [20], Chef [23], Salt [21] y Ansible [22], 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).

6. 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.

Agradecimientos

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

Referencias

1. Timeline of Virtualization Development, https://en.wikipedia.org/wiki/Timeline_of_virtualization_development
2. Operating-system-level virtualization https://en.wikipedia.org/wiki/Operating-system-level_virtualization
3. Google: 'EVERYTHING at Google runs in a container', https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/
4. DockerCon Video: Containerized Deployment at Facebook https://blog.docker.com/2014/07/dockercon-video-containerized-deployment-at-facebook/
5. Malte Schwarzkopf and Andy Konwinski and Michael Abd-El-Malek and John Wilkes: Omega: flexible, scalable schedulers for large compute clusters. SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic 351-364 (2013),
6. Abhishek Verma and Luis Pedrosa and Madhukar R. Korupolu and David Oppenheimer and Eric Tune and John Wilkes: Large-scale cluster management at Google with Borg. Proceedings of the European Conference on Computer Systems (EuroSys), (2015)
7. An update on container support on Google Cloud Platform https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html
8. The "Works on My MachineÇertification Program https://blog.codinghorror.com/the-works-on-my-machine-certification-program/
9. What is Docker? https://www.docker.com/what-docker
10. Use volumes https://docs.docker.com/engine/admin/volumes/volumes/
11. Docker container networking https://docs.docker.com/engine/userguide/networking/
12. Docker Architecture https://docs.docker.com/engine/docker-overview/#docker-architecture
13. Glossary https://docs.docker.com/engine/reference/glossary
14. Dockerfile reference https://docs.docker.com/engine/reference/builder/
15. docker build https://docs.docker.com/engine/reference/commandline/build/
16. docker run https://docs.docker.com/engine/reference/run/
17. Docker Compose https://docs.docker.com/compose/
18. Compose file version 3 reference https://docs.docker.com/compose/compose-file/
19. Preserve volume data when containers are created https://docs.docker.com/compose/overview/#preserve-volume-data-when-containers-are-created
20. Puppet https://puppet.com
21. Salt http://saltstack.com
22. Ansible https://www.ansible.com
23. Chef https://www.chef.io