UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

TREBALL DE FI DE GRAU

TÍTULO DEL TFG: ColoTe

TITULACIÓN: Grau en Enginyeria Telemàtica

AUTOR: Alfredo Varela Romero

DIRECTOR: Juan López Rubio

SUPERVISOR: Gerard Solé Castellví

FECHA: 30 de septiembre de 2016

Título: CoIoTe

Autor: Alfredo Varela Romero

Director: Juan López Rubio

Supervisor: Gerard Solé Castellví

Fecha: 30 de septiembre de 2016

Resumen

En los últimos años, el avance sobre los temas de IoT han llevado a un crecimiento muy elevado en la investigación tanto en dispositivos como de tecnologías para el cumplimiento de todas las barreras que nos ponían el software y el hardware que disponíamos hasta conseguir protocolos muy simples y sensores extremadamente pequeños y baratos. La demanda de este tipo de productos hace que muchas empresas cambie su orientación de mercado y se centren en el mundo de los sensores o incluso invierten en hacer una instalación de este tipo para mejorar su productividad.

Las investigaciones no solo se centran en crear nuevos protocolos o nuevas tecnologías sino también en reutilizar las que ya se conocen aplicando pequeños cambios o incluso sin mejoras ya que son completamente compatibles.

Por las razones expuestas, surge este proyecto para realizar un nuevo producto para una empresa la cual podrá disponer de toda sus infraestructura de manera remota y poder desplegarla en un lugar el cual el acceso o la conexión a internet sea escaso o nulo, hayan ocurrido desastres naturales... Y paralelamente la investigación de una nueva tecnología para desplegar aplicaciones utilizada sobretodo en el ámbito de servidores y serán llevadas al mundo de IoT para solucionar problemas como el alto consumo de ancho de banda al hacer una actualización del sistema o desplegar nuevos sistemas y el tiempo que conlleva realizar estas acciones.

En el proyecto se utilizara una Raspberry Pi para simular un nodo central o sumidero de datos en el cual mediante Docker se desplegará toda la infraestructura del core de la empresa AlterAid, aaaida. Se utilizarán sensores que mediante bluetooth low energy se conectaran a la Raspberry y se establecerá la red la cual permitirá el envío de los datos y su correcto almacenamiento para ser gestionados por la aplicacion.

Title : ColoTe

Author: Alfredo Varela Romero

Advisor: Juan López Rubio

Supervisor: Gerard Solé Castellví

Date: September 30, 2016

Overview

This document contains guidelines for writing your TFC/PFC. However, you should also take into consideration the standards established in the document Normativa del treball de fi de carrera (TFC) i del projecte de fi de carrera (PFC), paying special attention to the section Requeriments del treball, as this document has been approved by the EETAC Standing Committee

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

INTRODUCCIÓN

En los últimos años, el avance sobre los temas de IoT han llevado a crecimiento muy elevado en la investigación tanto como dispositivos como de tecnologías para el cumplimiento de todas las barreras que nos ponían el software y el hardware que disponíamos hasta conseguir protocolos muy simples y sensores extremadamente pequeños y baratos. La demanda de este tipo de productos hace que muchas empresas cambie su orientación de mercado y se centren en el mundo de los sensores o incluso invierten en hacer una instalación de este tipo para mejorar su productividad.

Por eso se decidió realizar este proyecto, el cual se llevará a cabo el despliegue de la plataforma aaaida en una Raspberry. La cual nos puede permitir el uso de dicha plataforma y sus aplicaciones en lugares a los cuales la conexión a internet es nula o escasa, hayan sufrido un desastre natural... ya que es una plataforma centrada en Ihealth que nos permite la monitorización del estado de un paciente. Gracias a las conexiones de la Raspberry podremos realizar una red con sensores que mediante Bluetooth se podrán comunicar y almacenar los datos.

Por otra parte el despliegue de la aplicación se llevará a cabo usando Docker, un software que permite contenerizar todo aquello que nuestra aplicaciones necesiten. El cual no sera muy util ya que nos permite instalar o actualizar contenedores por separado y no todo el sistema cada vez. Docker es utilizado básicamente en el mundo de servidores y no de IoT lo cual las ventajas que nos proporciona a la hora del despliegue son contrarrestadas con su complejidad en sistemas con capacidades limitadas.

Nuestra intención es poder desplegar toda la infraestructura de la empresa utilizando un sistema nuevo para IoT, comprobar su viabilidad y beneficios, ofreciendo un producto el cual podría ser de gran utilidad en caso de desastres como seria un sistemas de monitorización de pacientes.

La utilización de la Raspberry hace que esta infraestructura no sea extrapolable al cien por cien de los despliegues de IoT ya que sus especificaciones son bastantes superiores a las de sensores utilizados normalmente, pero como aproximación y prueba piloto para la tecnología nombrada anteriormente es suficiente. Docker podría abrirse un hueco en el mundo de IoT, aunque tenga este orientado a arquitecturas de servidor con recursos ilimitados, pero en dispositivos con recursos más limitados podría ser utilizado.

CAPÍTULO 1. VISIÓN GENERAL DEL PROYECTO

El propósito de este proyecto es desplegar toda la infraestructura de la empresa AlterAid en un dispositivo móvil en nuestro caso una Raspberry Pi, con la finalidad de poder conseguir desplegar las aplicaciones en lugares remotos, sin conexión a internet y/o que han sufrido un desastre natural.

Como segundo objetivo del proyecto queremos implementar un pequeño despliegue sensores haciendo que nuestra Raspberry sea el nodo central o sumidero de datos. Para poder llevar a cabo todo esto es necesario contar con la utilización de los contenedores usando la tecnología Docker. Puesto que éstos nos facilitaran el trabajo, su utilización fuera del mundo de servidores es motivo de investigación.

1.1. Docker

La idea detrás de Docker es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues.

Figura 1.1: Logotipo de Docker

1.2. Aaaida

Es una plataforma desarrollada por la empresa AlterAid la cual nos permite crearnos un usuario y con este usuario poder tener varios vínculos. Un Vínculo es un ente que representa aquello por el que el Usuario se ocupa y preocupa. Ejemplos de Vínculos de Usuario pueden ser familiares, pacientes, amigos, etc. Esta es una plataforma web la cual recoge todos los datos de las aplicaciones de la empresa y los almacena. Por eso la importancia de poder desplegar toda la infraestructura en un dispositivo que sea fácil de transportar y poder llegar a cualquier lugar.

Figura 1.2: Logotipo de Aaaida

1.3. Motivación

Conseguir el despliegue de toda la infraestructura de una empresa en un dispositivo de reducidas prestaciones y tamaño como seria una Raspberry Pipermitiría la apertura de nuevos campos de investigación y de mercado puesto que se podrían desarrollar otro tipo de aplicaciones para casos de emergencia o lugares con pocos medios. Utilizando una tecnología de servidores como es Docker conlleva un gran avance en el mundo de Internet of Things (de ahora en adelante IoT). El presente proyecto representa una prueba de concepto para versiones futuras en las cuales se podrían utilizar estas tecnologías para facilitar el despliegue tanto de las aplicaciones como de sus actualizaciones ya que al utilizar Docker solo se debería actualizar el contenedor independiente y no todo el sistema.

1.4. Objetivos

Los objetivos fijados para el desarrollo de este proyecto son los siguientes:

- Analizar y escoger las tecnologías a utilizar para el desarrollo del proyecto
- La instalación de Docker en una Raspberry Pi
- El despliegue de la infraestructura de la empresa AlterAid
- La creación de pequeños nodos
- El despliegue de la red
- Contemplar la viabilidad de creación de redes smart con estas tecnologías.

1.5. Organización del proyecto

En primer lugar, para poder cumplir todos los puntos de los objetivos necesitaremos estudiar un poco más a fondo todas las tecnologías que se utilizarán a lo largo de todo el proyecto. Serán explicadas en sus respectivos capítulos.

Como se ha explicado previamente se utilizará una Raspberry Pi, donde se desplegará la aplicación mediante Docker. Para poder llevarlo a cabo se tendrán que estudiar las diferentes posibilidades como, por ejemplo, si es posible ejecutar Docker en un sistema ARM o si se podrá virtualizar la Raspberry para poder hacer las pruebas de una manera más cómoda etc. Todo esto se podrá ver en el capítulo 2 y capítulo 3.

Una vez desplegada y ejecutada la aplicación es necesario arrancar aaaida. En el capítulo 4 se podrá ver una pequeña explicación del funcionamiento de la plataforma y sus funcionalidades.

Para terminar la parte técnica, vendría el paso de crear la red de sensores y comunicarnos con la Raspberry. Una vez se haya llevado a cabo esta conexión, se realizará el envío de los datos y su correcto almacenamiento. En el capítulo 5 se comentará como hacerlo para poder ser visualizados en la plataforma aaaida.

Por último, con la plataforma en la Raspberry y la red de sensores funcionando obtendremos el último capítulo donde se podrán ver las conclusiones y resultados sobre la viabilidad de este proyecto.

CAPÍTULO 2. VIRTUALIZACIÓN

2.1. ¿Qué es la virtualización?

La virtualización es la creación de una versión virtual (no física) de algo. Está basada en software, se puede aplicar a sistemas operativos, almacenamiento, servidores, aplicaciones, redes, etc. y es una manera de reducir gastos y aumentar eficiencia y agilidad en las empresas.

2.1.1. Tipos de virtualización

Estos son los 4 tipos de virtualización más habituales.

2.1.1.1. Virtualización de servidores

La virtualización en servidores ayuda a evitar ineficiencias ya que permite ejecutar varios sistemas operativos en una máquina física con máquinas virtuales con acceso a los recursos de todos. También permite generar un clúster de servidores en un único recurso para así mejorar mucho más la eficiencia y la reducción de costes. También permite el aumento de rendimiento de las aplicaciones, la disponibilidad al aumentar la velocidad en la carga de trabajo.

2.1.1.2. Virtualización de escritorios

La implementación de escritorios virtualizados permite ofrecer a las sucursales o empleados externos de forma rápida y sencilla un entorno de trabajo y una reducción de la inversión a la hora de gestionar cambios en éstos.

2.1.1.3. Virtualización de red

Se trata de reproducir una red física completa mediante software para poder ejecutar los mismos servicios que una red convencional y sus dispositivos. Cuentan con las mismas características y garantías que las redes físicas con las ventajas que nos ofrece la virtualización además de la liberación del hardware.

2.1.1.4. Almacenamiento definido por software

La virtualización del almacenamiento permite prescindir de los discos de los servidores. Los combina en depósitos de almacenamiento de alto rendimiento y los distribuye como software. Este nuevo modelo permite aumentar la eficiencia en el guardado de datos.

2.1.2. Ventajas de la virtualización

Como se ha podido apreciar en los tipos de virtualización presentados anteriormente, ésta conlleva una mejora considerable tanto en el rendimiento, agilidad, flexibilidad, escalabilidad, etc. como en una reducción considerable de los costes económicos y de tiempo y una simplificación en la gestión de la infraestructura.

- Reduce los costes de capital y los gastos operativos.
- Minimiza o elimina los tiempos de inactividad.
- Aumenta la productividad, la eficiencia, la agilidad y la capacidad de respuesta.
- Implementa aplicaciones y recursos con más rapidez.
- Garantiza la continuidad del negocio y la recuperación ante desastres.
- Simplifica la gestión del centro de datos.

2.2. ¿Qué es Docker?

Docker es un proyecto Open Source basado en contenedores de Linux. Es básicamente un motor de contenedores que usa características del Kernel de Linux.

La idea detrás de Docker es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues.

De una manera más sencilla, Docker proporciona la opción de introducir en pequeños contenedores todo aquello que nuestra aplicación necesite. Permite desplegarla en cualquier máquina que tenga Docker instalado, sin preocuparnos de nada más.

Se podría decir que Docker son pequeñas “máquinas virtuales” pero muchos más ligeras ya que utilizan el sistema operativo de donde se ejecuta y el contenido relevante para ejecutar la aplicación está dentro de los contenedores.

Docker es:

- Open-Source para la gestión de ”virtualización de contenedores”
- Aísla múltiples sistemas de archivos en el interior del mismo host
  - Las instancias se llaman Contenedores
  - Te dan la ilusión de estar dentro de una máquina virtual
- Piensa en entornos de ejecución o ”sandboxes”
- No hay necesidad de un hypervisor (rápido de ejecutar)
- Requiere x64 y Linux kernel 3.8+

Docker no es:

- Un lenguaje de programación
- Un sistema operativo
- Una máquina virtual
- Una imagen en el concepto tradicional de la máquina virtual basada en hipervisor

2.3. Máquinas Virtuales vs Docker

Las máquinas virtuales incluyen toda la aplicación, los binarios y librerías necesarias y todo un sistema operativo. Esto implica que ocupen mucho espacio, que el tiempo de ejecución sea lento y la necesidad de un Hipervisor para su utilización.

Por lo contrario, los Docker container incluyen la aplicación y todas sus dependencias pero comparten el núcleo con otros contenedores, funcionando como procesos aislados en el sistema de ficheros del sistema operativo. Los Docker container no están vinculados a ninguna infraestructura específica: se ejecutan en cualquier ordenador, en cualquier infraestructura y en cualquier cloud.

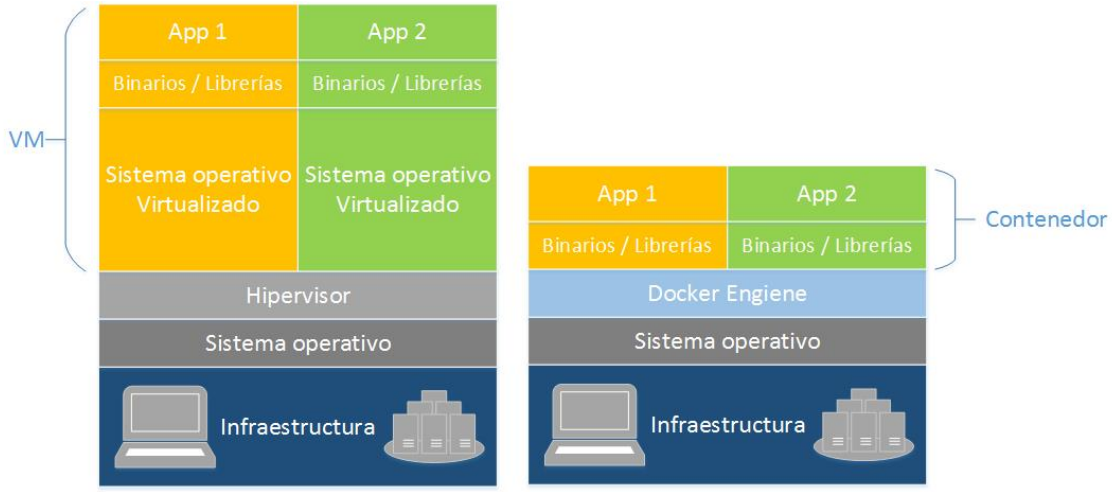


Figura 2.1: Comparativa de máquina virtual y Docker

En la figura 2.1 se aprecian las diferencias comentadas anteriormente. La primera columna corresponde a la virtualización de 2 aplicaciones mediante la capa intermedia del Hipervisor, el cual nos permite ejecutar los sistemas operativos virtualizados. Estos, a su vez, permiten ejecutar todos los binarios y librerías que requiere cada aplicación y, finalmente, la aplicación. La segunda columna corresponde a la contenerización de 2 aplicaciones mediante Docker Engine. Podemos observar que no hace falta un hipervisor ya que utilizan el sistema operativo de la infraestructura de donde son ejecutadas, pero sí son necesarios los binarios y las librerías. A simple vista se puede apreciar que, si eliminamos el sistema operativo, la infraestructura completa se hace más liviana y rápida de ejecutar.

2.4. ¿Por qué Docker?

Según lo listado en los apartados anteriores, se decidió utilizar Docker para realizar el despliegue de las aplicaciones por:

Cumplir con la condición de tecnología que pueda ser almacenada en dispositivos con una memoria reducida.

Su rapidez a la hora de levantar el servicio. En el mundo de IoT el tiempo es un bien preciado y los sistemas se pueden apagar y encender constantemente para evitar gastos de energía innecesarios.

La facilidad para poder desplegar los servicios. Con Docker desplegar servicios es muy sencillo, solo requiere tenerlo instalado y ejecutar el contenedor para que el servicio esté activo.

La sencillez a la hora de mantener el sistema. Si hay que hacer actualizaciones o controles de una pequeña parte del servicio solo habría que cambiar o actualizar ese contenedor y no todo el sistema. Esto es un gran ahorro en recursos y tiempo.

Por todos estos motivos se piensa que Docker puede ser una buena tecnología para desplegar sistemas de Iot. También hay que tener en cuenta que, aunque la Raspberry 3 no es un aparato con unas capacidades limitadas como podría ser un sensor utilizado normalmente, sí que cumple con todas las funciones listadas anteriormente y, por lo tanto, puede servir como preámbulo para la utilización en el resto de despliegues.

2.5. ¿Cómo funciona Docker?

En este punto se explicarán brevemente aspectos de la arquitectura, funcionamiento y puesta en marcha de Docker.

2.5.1. Arquitectura

Docker usa una arquitectura cliente-servidor. El cliente de Docker se comunica con el Daemon de Docker para crear, ejecutar y distribuir los contenedores. Tanto el cliente como el Daemon pueden estar en el mismo sistema o pueden conectarse remotamente. Como Docker usa el kernel de Linux para su ejecución, si el sistema operativo del sistema no es éste, se deberá usar una pequeña capa extra en la arquitectura de tipo VM (boot Docker) para poder correr Docker en la máquina.

2.5.1.1. Cliente de Docker

Es la principal interfaz de usuario para Docker. Acepta los comandos del usuario y se comunica con el Daemon de Docker.

2.5.1.2. Imágenes de Docker (Docker Images)

Las imágenes de Docker son plantillas de sólo lectura, que nos permitirán crear contenedores basados en su configuración.

2.5.1.3. Registros de Docker (Docker Registries)

Los registros de Docker guardan las imágenes. Éstos son repositorios públicos o privados donde se pueden subir o descargar imágenes. Sería similar a GitHub para imágenes de Docker (Docker Hub).

2.5.1.4. Contenedores de Docker (Docker Containers)

El contenedor de Docker contiene todo lo necesario para ejecutar una aplicación. Cada contenedor se crea de una imagen de Docker y es una plataforma aislada.

En la figura 3.1 podemos apreciar gráficamente cómo sería la arquitectura básica. El cliente podría hacer los comandos básicos de docker:

`Docker Build`: Hacer un build de un DockerFile y generar una imagen de Docker. En la figura está representado siguiendo las flechas rojas en las cuales podemos ver que el cliente se comunica con el Daemon y éste genera la imagen.

`Docker Pull`: Permite descargar una imagen de los repositorios de Docker. En la imagen se puede observar siguiendo las flechas verdes e, igual que el comando anterior, se comunica el cliente con el Daemon para que éste proceda a hacer la descarga de la imagen de mongoDB del repositorio.

`Docker Run`: Ejecuta una imagen para generar un contendor de ésta. Siguiendo las flechas azules, veremos cómo nuevamente el cliente, al ejecutar esa comanda, se comunica con el Daemon. Este busca la imagen que se quiere ejecutar, en estecaso la imagen de Ubuntu y se levanta un contenedor con la configuración de la imagen.

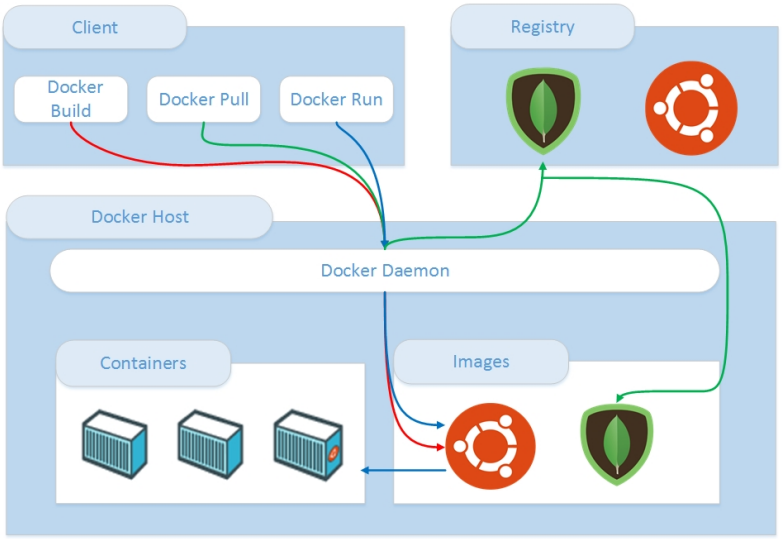


Figura 2.2: Infraestructura de Docker

2.5.2. Creación de Imágenes

Como se comenta en el punto anterior, las imágenes de Docker son las plantillas para poder levantar los contenedores. Por eso la importancia de saber crear imágenes y personalizarlas ya que sólo permiten lectura y los cambios que hagamos en los contenedores no se verán reflejados en éstas.

La manera más sencilla de crear una imagen es descargarla del Docker Hub con el comando explicado anteriormente:

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Este comando nos permite descargar una imagen en una versión concreta o tag dependiendo de nuestras necesidades. Por defecto, si no se pone nada, descargará la última.

```
alfredo@alfredo:~$ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo

8ad8b3f87b37: Pull complete
5947be99d359: Pull complete
d5a4577c6007: Pull complete
acb97586a200: Pull complete
d11260d069a3: Pull complete
bf102d35e390: Pull complete
f4964f6a9bfa: Pull complete
8b392ba3e8bf: Pull complete
c023b73abe56: Pull complete
Digest: sha256:8ff7bd4acdb123e3922a7fae7f73efa35fba35af33fad0de946ea31370a23cc4
Status: Downloaded newer image for mongo:latest
alfredo@alfredo:~$
```

(a) Docker pull

```
alfredo@alfredo:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mongo               latest              48b8b08dca4d        2 weeks ago         366.4 MB
alfredo@alfredo:~$
```

(b) Docker images

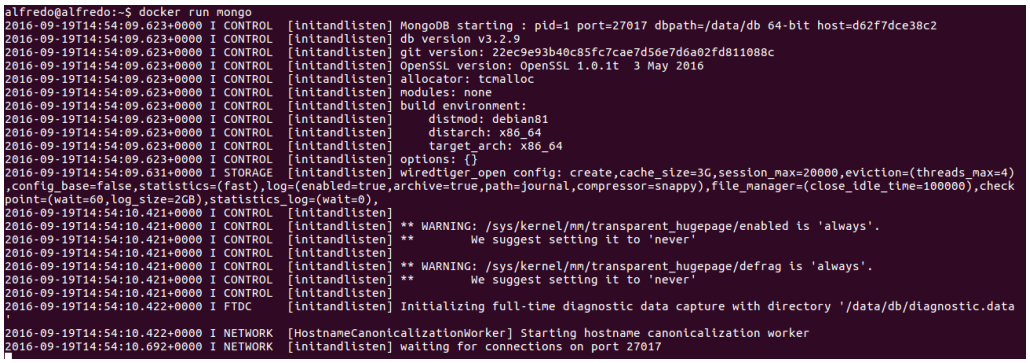
Figura 2.3: Comandos de docker (I)

En las figuras 2.3 podemos apreciar como se descarga la última versión de la imagen de MongoDB y nos genera la imagen.

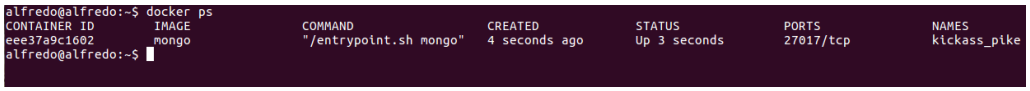
Una vez obtenida la imagen se pasará a levantar el contenedor para poder ejecutar el servicio con otro de los comandos explicados.

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

(a) Docker ps



(b) Docker run



(c) Docker ps

Figura 2.4: Comandos de docker (II)

En las figuras 2.4 se puede apreciar cómo utilizando el comando `docker ps` que permite ver qué contenedores están levantados, no hay ninguno (a) y cómo al inicializar con el comando `docker run mongo` evanta el servicio (b) y esta vez sí aparece el contenedor (c).

La segunda manera de crear y personalizar imágenes es mediante un `DockerFile`, que es un documento de texto donde se encuentran los comandos que se deben ejecutar para generar nuestra imagen.

El comando `docker build` comunica al Daemon de Docker que debe de leer el `DockerFile` del directorio actual y seguir las instrucciones línea por línea para la creación de nuestra imagen.Este proceso va pintando los resultados por pantalla y generando imágenes intermedias para obtener así una caché que nos permitirá en caso de errores, una vez corregido el DockerFile, continuar desde el punto conflictivo.

```
FROM docker/whalesay:latest
CMD echo "Proyecto CoIoTe" | cowsay
```

Aquí tenemos un ejemplo sencillo de DockerFile que nos servirá para explicar de una manera rápida como crearlos.

`FROM` indica la imágen base que va a utilizar para seguir futuras instrucciones. Buscará si la imagen se encuentra localmente, en caso de que no, la descargará de internet. En nuestro ejemplo utiliza la última versión de la imagen docker/whalesay.

La instrucción `CMD` solo puede aparecer una vez en un DockerFile, si colocamos más de uno, solo el último tendrá efecto. El objetivo de esta instrucción es proveer valores por defecto a un contenedor. Estos valores pueden incluir un ejecutable u omitir un ejecutable que en dado caso se debe especificar un punto de entrada o entrypoint en las instruccio-

nes. En nuestro caso pintaremos un texto que se le pasara a la aplicación cowsay.

Existen muchas más instrucciones, alguna de estas serán explicadas más adelante cuando sea necesario su uso.

Una vez ejecutado el comando `docker build -t docker-whale .` donde el `-t` nos permite darle nombre a la imagen y el `.` encontrar el DockerFile para ser compilado. Como se ve en la Figura 2.5 descarga del repositorio la imagen ya que no la tenemos en local creará una imagen intermedia que nos proporciona la caché en caso de fallo, continuará con la siguiente orden creando una nueva imagen nueva borrando las anteriores hasta obtener la imagen definitiva.

```
alfredo@alfredo:~/mydockerbuild$ sudo docker build -t docker-whale .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM docker/whalesay:latest
latest: Pulling from docker/whalesay
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
 ---> 6b362a9f73eb
Step 2 : CMD echo "Proyecto CoIoTe" | cowsay
 ---> Running in 598204c0e3a3
 ---> ac80256777e9
Removing intermediate container 598204c0e3a3
Successfully built ac80256777e9
```

Figura 2.5: Build del DockerFile

Una vez hecho el build del DockerFile podemos comprobar que nuestra imagen está creada correctamente (Figura 2.6) y pasaremos a levantar el contenedor. Una vez levantado nos saldrá por pantalla el icono de docker “diciendo” la frase que le indicamos en el fichero. (Figura 2.7)

```
Successfully built ac80256777e9
alfredo@alfredo:~/mydockerbuild$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker-whale        latest              ac80256777e9        About a minute ago   247 MB
mongo               latest              48b8b08dca4d        2 weeks ago          366.4 MB
docker/whalesay     latest              6b362a9f73eb        16 months ago        247 MB
alfredo@alfredo:~/mydockerbuild$
```

Figura 2.6: Docker images

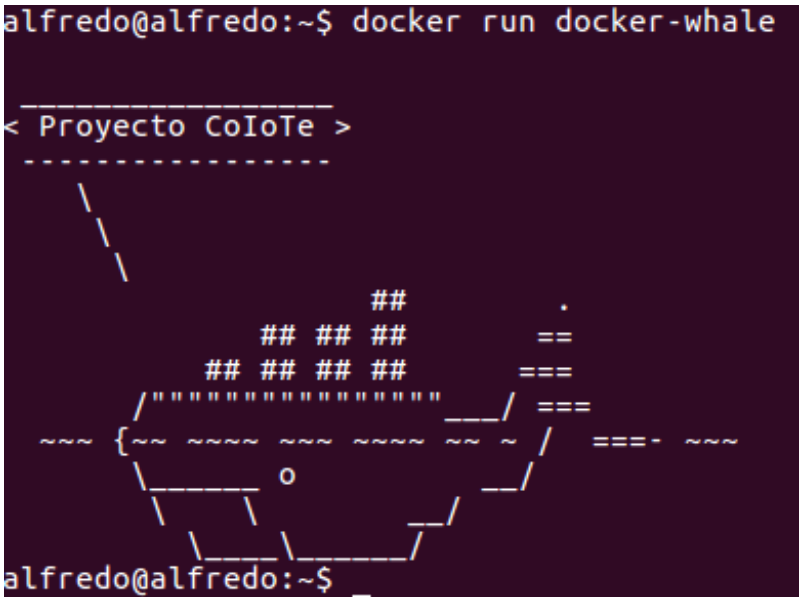


Figura 2.7: Docker run docker-whale]

2.5.3. Docker Compose

Docker Compose es un orquestador que nos permite ejecutar aplicaciones que utilicen varios contenedores a la vez. Se creará un archivo docker-compose.yml donde se configurarán todos los servicios necesarios para nuestra aplicación. Una vez ejecutado este archivo nos generará todas las imágenes y con éstas los contenedores especificados a la vez que arrancará la aplicación.

Los comandos que se utilizarán serán similares a los utilizados en la creación de imágenes.

```
docker-compose up
```

Permite ejecutar el servicio y levantar todos los contenedores.

```
docker-compose stop
```

Detiene el servicio y detiene los contenedores.

Docker-compose también permite ejecutar solo partes del.yml para levantar y/o detener solo algunos de los contenedores, pasándoles por parámetro el nombre del contenedor.

```
version: '2'
services:
 mongo:
  container_name: mongo
  restart: always
  image: partlab/ubuntu-arm-mongodb
  volumes:
   - mongo-data:/data/db
  command: /usr/bin/mongod --smallfiles --journal
```

```
 aaaida:
  container_name: aaaidaArm
  restart: always
  image: alteraid/aaaida-datastore-arm
  links:
   - mongo:mongo
  environment:
   - NODE_ENV=docker
  ports:
   - "40000:40000"
volumes:\newline
 mongo-data:
   driver: local
```

Este es un ejemplo de docker-compose, exactamente el utilizado en el proyecto para poder levantar la aplicación aaaida en la Raspberry. La importancia en los espacios es imprescindible para que este funcione ya que da la jerarquía adecuada para que docker-compose lo entienda. Ahora pasaremos a explicar brevemente los argumentos que podemos encontrar en el. En primer lugar tenemos los servicios, en nuestro caso son 2, una base de datos MongoDB y aaaida.

Dentro de mongo tendremos los siguientes:

`container name`: Le da un nombre al contenedor para no tener que referirse a él por la id o el nombre aleatorio que proporciona Docker.

`restart`: Nos permite que en caso de fallida o reinicio del sistema este contenedor vuelva a ejecutarse.

`image`: Para poder especificar la imagen que utilizaremos para este servicio.

`volumes`: En caso de mongo necesita un directorio donde almacenar los datos, esto especifica donde se creará el volumen de datos.

`command`: El comando que le pasamos al contenedor al momento de su ejecución. En este caso concreto es muy importante ejecutar esta comanda ya que activará el journaling, que por defecto viene desactivado.

En el siguiente servicio, el de aaaida, a parte de disponer los mismo argumentos basicos como serian `container name`, `restart` o `image` encontraremos alguno más como:

`links`: Define el enlace entre contenedores.

`environment`: Para poder poder pasar variables.

`ports`: Define el mapeo de puertos.

Para finalizar el archivo se encuentra los `volumes` donde generamos el volumen que hemos especificado dentro del servicio de mongo.

En puntos posteriores se explicará más detenidamente el funcionamiento y la puesta en marcha del Docker Compose ya que el archivo utilizado para el ejemplo es el utilizado en la Raspberry.

2.6. Conclusiones

Como se puede observar, al virtualizar los recursos disponibles obtenemos un aumento en rendimiento y reducción de los costo.

Docker nos permite el despliegue de aplicaciones de una manera sencilla y en cualquier plataforma que lo soporte.

Las diferencia más significativa entre las máquinas virtuales y Docker sería el uso del hipervisor y la necesidad de un sistema operativo para las máquinas virtuales. Cosa que en Docker no es necesario, esto hace que las aplicaciones desplegadas de esta manera tengan una reducción de tiempo al ejecutarse y en de tamaño.

CAPÍTULO 3. RASPBERRY PI

3.1. ¿Por qué Raspberry Pi?

La utilización de una Raspberry Pi en este proyecto son claras y comentadas con anterioridad. Es un dispositivo de un tamaño muy reducido y portátil, que nos facilita mucho el poder llevarla a cualquier lugar sin dificultades y poder ofrecer un servicio en cualquier lugar. Tiene unas prestaciones más que aceptables para un dispositivo de ese tamaño, conexiones inalámbricas como serian Wifi y Bluetooth integrados cosa impredecible para las comunicaciones con los sensores. Todo a un precio muy atractivo el cual permite su compra.

Algunas de las especificaciones mas importantes de las Raspberry Pi 3 son las siguientes:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1 y Bluetooth Low Energy (BLE)
- 1GB RAM
- Ethernet port
- Micro SD card slot

Figura 3.1: Raspberry Pi 3

3.2. Virtualización de la Raspberry

Al inicio del proyecto no se disponía de una Raspberry para poder realizar las pruebas y se decidió emular la mediante qemu. Qemu es una aplicación que nos permite emular mediante máquinas virtuales gran parte de los sistemas operativos. Lo cual es realmente sencillo emular una imagen de Raspbian si no fuera por un detalle, como se comenta en el apartado ¿Qué es Docker? Docker necesita un sistema x64 o Linux kernel 3.8+ y Raspberry ejecuta un sistema ARM lo que conlleva que Docker no sea compatible a primera instancia con la Raspberry. La solución para este gran problema es la utilización de Hypriot, una imagen de Raspbian modificada con Docker instalado. La instalación de esta imagen requiere hacer unos retoques ya que no es del todo compatible el kernel de qemu con el de la imagen de Hypriot. Cosa que aun que permite emular perfectamente la imagen con Docker, no nos permite el uso correcto. Por lo tanto se decidió apartar por completo la posibilidad de poder emular una Raspberry con Docker instalado ya que las incompatibilidades del kernel no lo permitían.

En el apéndice se podrá ver los pasos a seguir para la instalación y emulación de la imagen.

3.3. Raspberry y Docker

Después de descartar completamente la opción de emular la imagen de la Raspberry se decidió probar en la Raspberry que disponía la empresa la imagen corria perfectamente. Y si, la imagen iba perfectamente y ejecutaba Docker sin ningún problema. Ahora el problema era la Raspberry, era el primer modelo el cual era viejo y no disponía de módulos para conexiones inalámbricas integrados por lo que se decidió aprovechando el lanzamiento de la nueva Raspberry 3 comprarla ya que si dispone de de conexiones inalámbrica.

3.3.1. Instalación de la imagen en la Raspberry

La como se ha dicho, la imagen a la imagen de Raspbian no es posible instalar Docker de manera como lo haríamos en un sistema operativo como linux, es necesario una imagen la cual ya lo tenga instalado en nuestro caso utilizamos la imagen de Hypriot. La instalación se llevó a cabo de esta manera:

```
$ curl -sSL http://downloads.hypriot.com/docker-hypriot_1.8.2-1_armhf.deb
>/tmp/docker-hypriot_1.8.2-1_armhf.deb
$ sudo dpkg -i /tmp/docker-hypriot_1.8.2-1_armhf.deb
$ rm -f /tmp/docker-hypriot_1.8.2-1_armhf.deb
$ sudo sh -c 'usermod -aG docker $SUDO_USER'
$ sudo systemctl enable docker.service
```

- Primero descarga la imagen deseada y la montara en un directorio temporal.
- Instalará los paquetes con el comando dpkg.
- Borra el archivo comprimido.

- Ejecutará el comando donde añade el usuario a un grupo docker.
- Pondrá en marcha el Daemon de Docker.

Una vez tenemos la imagen preparada podremos hacer una copia binaria en una SD y ya estará lista para poder ser arrancada desde una Raspberry.

3.3.2. Creación de las imágenes de Docker

Una vez tenemos la Raspberry lista, se pasará a la creación de las imágenes Docker para poder desplegar Aaaida. En primer lugar se necesita un contenedor que contenga una base de datos en este caso, MongoDB. Para la creación de la imagen utilizaremos ya una creada y que esté en los repositorios de Docker Hub. Se necesita una imagen que sea compatible con ARM cosa que no resulta sencillo ya que la gran mayoría de las imágenes no están pensado para esta arquitectura y los pocos disponibles no estaban operativos. Otra restricción que teníamos era que debía ser una base de datos sin la necesidad de ingresar un usuario y su contraseña, cosa que no es habitual pero resultó una obligación en algunas imágenes. Por lo tanto se utilizó la siguiente imagen de Docker Hub:

`partlab/ubuntu-arm-mongodb`

La otra imagen que se utilizar será la propia de Aaaida, pero igual que la imagen de mongo debería ser compatible con una arquitectura ARM y al ser una aplicación propia de la empresa no se encontraría en repositorios público, tendríamos que crearla nosotros mediante un Dockerfile. En la siguiente figura se puede ver el Dockerfile que se utilizara para poder crear el contenedor con Aaaida.

```
FROM ioft/armhf-debian
RUN apt-get update; apt-get -y install curl
RUN set -ex \
&& for key in \
9554F04D7259F04124DE6B476D5A82AC7E37093B \
        94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
        0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
        FD3A5288F042B6850C66B31F09FE44734EB7990E \
        71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
        DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
        B9AE9905FFD7803F25714661B63B535A4C206CA9 \
        C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
;  do \
gpg --keyserver ha.pool.sks-keyservers.net --recv-keys "$key"; \
done

ENV NODE_VERSION 4.4.5

RUN curl -SLO
"https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-
linux-armv7l.tar.gz" \
```

```
&& curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
&& gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
&& grep "node-v$NODE_VERSION-linux-armv7l.tar.gz\$" SHASUMS256.txt |
sha256sum -c - \
&& tar -xzf "node-v$NODE_VERSION-linux-armv7l.tar.gz" -C /usr/local
--strip-components=1 \
&& rm "node-v$NODE_VERSION-linux-armv7l.tar.gz" SHASUMS256.txt.asc
SHASUMS256.txt

COPY scripts/rpi_docker/entrypoint /entrypoint
RUN mkdir /aaaida
WORKDIR /aaaida
CMD ["/entrypoint"]
EXPOSE 40000
COPY . /aaaida
```

Al principio del Dockerfile se ve la acción `FROM` que ya fue explicada en la cual carga una imagen de una debian para ARM, como observación se puede detectar que la imagen del mongo es para Ubuntu ARM y la imagen base para la aplicación de Aaaida es una debian ARM, distribuciones de linux diferentes en los dos contenedores. Esto demuestra que cada contenedor es un ente aislado que no tiene que depender del resto de contenedores.

A continuación de declarar la imagen base, vienen una serie de acciones con la instrucción `RUN`, que nos permite ejecutar cualquier comando. En el primer caso que hace un update e instala el curl.

Las siguientes tres instrucciones de las cuales dos de ellas son un `RUN` y la restante un `ENV`, que configura las variables de entorno, son para poder instalar el Node.js en el contenedor para poder compilar el código de Aaaida.

Una vez instalado se utiliza la acción `COPY`, que como su nombre indica, copiara el contenido de un directorio en un directorio del contenedor.

Si seguimos se ve que crea un directorio con el nombre de `aaaida` y hace que ese directorio sea el directorio de trabajo con la instrucción `WORKDIR`.

Después Ejecutará `[/entrypoint"]` que al escribirlo en formato JSON la instrucción `CMD` ejecuta el contenido sin shell. El contenido de este script especifica los puertos, el host... de la aplicación. Se podrá ver su contenido en el apéndice.

Para terminar con las 2 últimas instrucciones, que son `EXPOSE` y `COPY`, la cual la primera indica los puertos que el contenedor tendrá activos y por los cuales escuchara. El segundo hará una copia desde el punto raíz donde se ejecute el Dockerfile en el directorio creado anteriormente de `aaaida`.

CONCLUSIONES

Escriure aquí les conclusions del projecte.

BIBLIOGRAFÍA

[1] Cognoms-autor, Inicial-nom. “Títol del capítol”. *Títol del llibre*. (Editor. Ciutat. Any publicació): pagina1–paginaN.

[2] Cognoms-autor, Inicial-nom. “Títol de l’article”. *Títol de la revista*. **volum**(numero), pagina1–paginaN. (Any publicació)

APÉNDICES

APÉNDICE A. EXEMPLE DE PROVA D'UN APÈNDIX

Text de prova