Seminario 04. Tecnologías para el despliegue

TECNOLOGÍAS DE LOS SISTEMAS DE INFORMACIÓN EN LA RED, 2016-17

El material de este seminario se compone de una parte descriptiva con ejemplos, y otra en inglés, en un documento separado[1], que reúne manuales de referencia sobre Docker, abordando tanto las opciones de las órdenes como las directivas de los archivos de configuración.

Objetivos del seminario 4

- Aplicar varios conceptos introducidos en el tema "Despliegue de Servicios", con un nivel de detalle suficiente para comprender su alcance
- Introducir algunos casos en sintonía con las tecnologías actuales
- Ilustrar un ejemplo que incluya todos los pasos necesarios para llegar al despliegue final
- Cubrir los prerrequisitos necesarios para la última actividad de laboratorio.

CONTENIDO

[1] *Documentación adicional sobre Docker*

1 INTRODUCCIÓN

En el tema "Despliegue de Servicios" se presentaron:

- Un modelo de aplicación distribuida integrada por componentes autónomos
- Múltiples aspectos a contemplar en el despliegue, destacando:
  - La especificación y configuración de los componentes
  - El descriptor de despliegue
- Un ejemplo de despliegue (que conviene repasar)
- La problemática del destino del despliegue, introduciendo conceptos y modalidades relacionados con la Computación en la Nube (CC), así como un caso concreto (Windows Azure)

Este seminario pretende afianzar estos conceptos mediante un caso de despliegue, con suficientes detalles para realizarlo en el laboratorio. Las tecnologías concretas que intervienen son:

- Aplicaciones (en NodeJS) que se comunican mediante mensajería (ØMQ)
- Otro software adicional que puedan requerir las aplicaciones
- Sistemas (LINUX) sobre los que se ejecutan las aplicaciones con sus requisitos

Estas piezas (aplicación + requisitos + sistema) se reúnen formando un componente
La tecnología que pretendemos emplear ha alcanzado un punto en el que existe un consenso acerca de su utilidad y aplicabilidad, pero las implementaciones que encontramos pueden ser inestables dada la velocidad de evolución actual. Es especialmente necesario señalar que la documentación, características y ejemplos pueden estar muy ligados a la versión de la implementación.
En el caso de Docker, que es una implementación de la tecnología de contenedores, hemos intentado que todo el material e informaciones recogidas sean compatibles con la **versión 1.12**, dado que se trata de un punto crucial en dicha implementación. Es fácil encontrar documentación y ejemplos incompatibles con esta versión. Deberemos fijarnos en que:

- Cuando se menciona swarm, se indique como *docker en modo swarm*, y **nunca** como *swarm-kit*
- Cuando se menciona compose, se especifique la **versión 2** en el fichero de configuración docker-compose.yml

Estas precauciones pueden no ser suficientes, pero al menos deben manteneros alerta ante esta posible fuente de errores.

1.1 Aprovisionamiento

Llamamos **aprovisionamiento** (*provisioning*) a la tarea de reservar la infraestructura necesaria para una aplicación distribuida

- Reservar recursos específicos para cada instancia de componente (procesador + memoria + almacenamiento)
- Reservar recursos para la intercomunicación entre componentes.

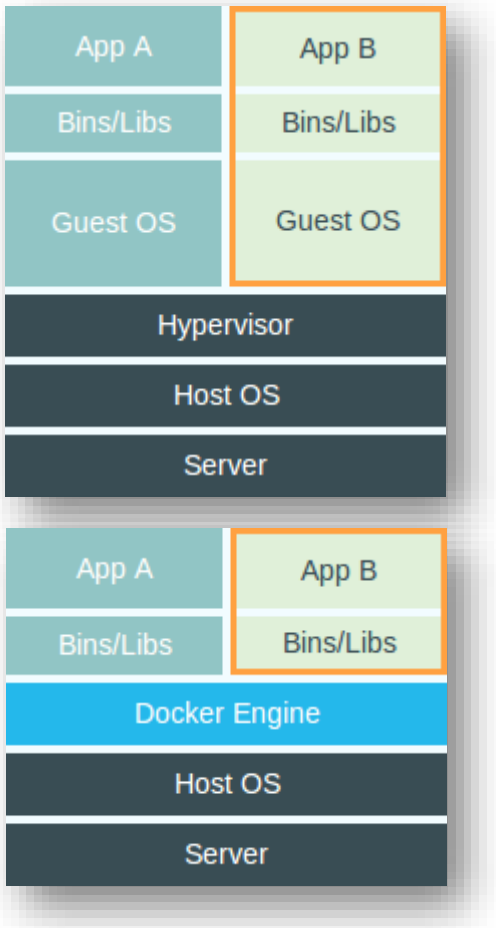
La infraestructura suele concretarse en un *pool* de **máquinas virtuales** interconectadas.

- El componente y sus requisitos se implementan y ejecutan sobre una máquina virtual

En nuestro caso optamos por una versión *ligera*[2] de máquina virtual, denominada **contenedor**

- El sistema del huésped coincide con el del anfitrión, por lo que no se requiere duplicarlo ni emularlo
- El anfitrión ha de disponer de un software de *contenerización*, que ofrece ciertos servicios de aislamiento y replicación

Cada componente se implementa sobre un contenedor



1.2 Configuración de componentes

Para cada componente hay una especificación de su configuración que incluye

- El software a ejecutar
- Las dependencias que deberán ser concretadas

Nuestra decisión anterior acerca de los contenedores supone que el software del componente…

- Debe ser compatible con el S.O. del anfitrión
- Puede completarse con algún software adicional
- Debe configurarse e inicializarse

Estas tareas serán responsabilidad de la propia operación de creación de la imagen, y del fichero de propiedades del contenedor

1.3 Plan de despliegue

Tal y como se describió en el apartado 3 del tema "Despliegue de Servicios", la lista de acciones a ejecutar para llevar a cabo el despliegue viene especificada como un algoritmo o plan

- En caso de una herramienta automatizada, existirá una aplicación que interpretará (*orquestación*) la especificación, y llevará a cabo las acciones
- En su ausencia, se desarrollará un programa a medida de las indicaciones del plan de despliegue

El despliegue manual se reserva para pruebas y casos sencillos

- Es inmanejable para una aplicación distribuida de tamaño medio.

[2] Ver segunda ilustración

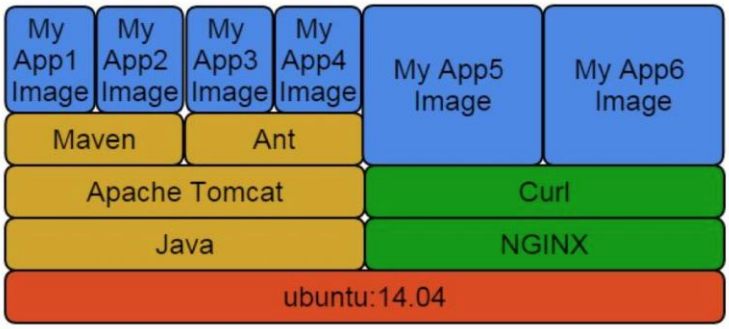
2 DE LAS MÁQUINAS VIRTUALES A LOS CONTENEDORES

Actualmente las tecnologías de virtualización han permitido implementar servicios con flexibilidad, pero éstos deben rentabilizar una instalación completa. P.ej., una instalación virtualizada mínima puede consumir 500MB

- No es práctico tomarla como base para un servicio que devuelva la hora actual (*demasiado equipaje*)
- Sí que es rentable como base para un servicio de correo electrónico con una decena de usuarios; por tanto, es adecuada para servicios relativamente pesados

Las implementaciones actuales en tecnología de contenedores, como Docker, son suficientemente maduras para plantearse como alternativa a las máquinas virtuales

- Ventaja: suponiendo que una imagen de 1GB usada por 100MVs consuma 100GBs…
  - Si hay 900MBs inmutables, 100 contenedores Docker consumirían 0,9+0,1*100=10,9GBs -> reduce **espacio**
    - Los contenedores consumen aproximadamente entre 10 y 100 veces menos recursos que sus equivalentes virtuales
  - Si la parte inmutable se encuentra "precargada", nos ahorramos ese tiempo (90%) para iniciar cada instancia Docker -> reduce **tiempo**
- Inconvenientes (como cualquier sistema de contenedores)
  - Menos flexible que las MVs
  - El aislamiento imperfecto entre contenedores puede provocar interferencias y problemas de seguridad



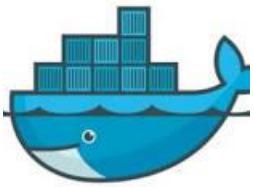
Con algunas simplificaciones, el sistema basado en contenedores ilustrado a la derecha nos permite detallar el ahorro respecto a un sistema basado en máquinas virtuales: el equivalente virtualizado daría lugar a **una MV completa por cada imagen de aplicación** (6 en total), incluyendo (de izquierda a derecha)…

1. Ubuntu+Java+Tomcat+Maven+My App1 Image
2. Ubuntu+Java+Tomcat+Maven+My App2 Image
3. Ubuntu+Java+Tomcat+Ant+My App3 Image
4. Ubuntu+Java+Tomcat+Ant+My App4 Image
5. Ubuntu+NGINX+Curl+My App5 Image
6. Ubuntu+NGINX+Curl+My App6 Image

Sumando: 6*Ubuntu+4*Java+4*Tomcat+2*Maven+2*Ant+2*NGINX+2*Curl+ My AppX Image (0<X<7)… **¡La diferencia es MUY sustancial!**

La tecnología de contenedores es tan ligera que posibilita la virtualización **de una aplicación**

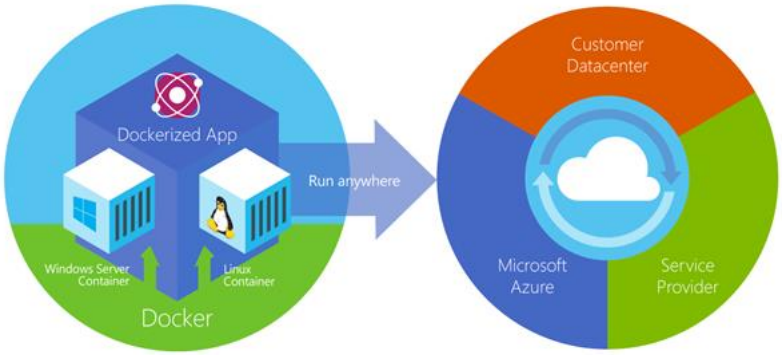
3 INTRODUCCIÓN A DOCKER



Docker ofrece un API para ejecutar procesos de forma aislada, por lo permite construir una PaaS

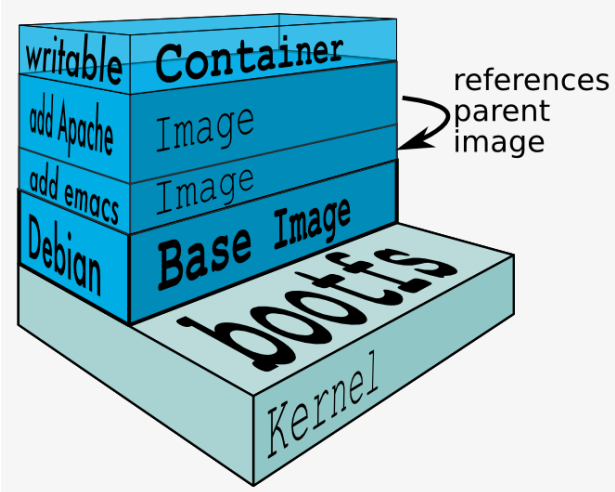
La implementación de Docker se basaba originalmente en LXC, pero posteriormente esa dependencia ha sido suavizada para interactuar con otros implementadores de contenedores. En una segunda etapa se utilizó *libcontainer*

- Actualmente *runc*
- Junto a Microsoft se ha conseguido una implementación nativa para Windows



Respecto a LXC, Docker añade:

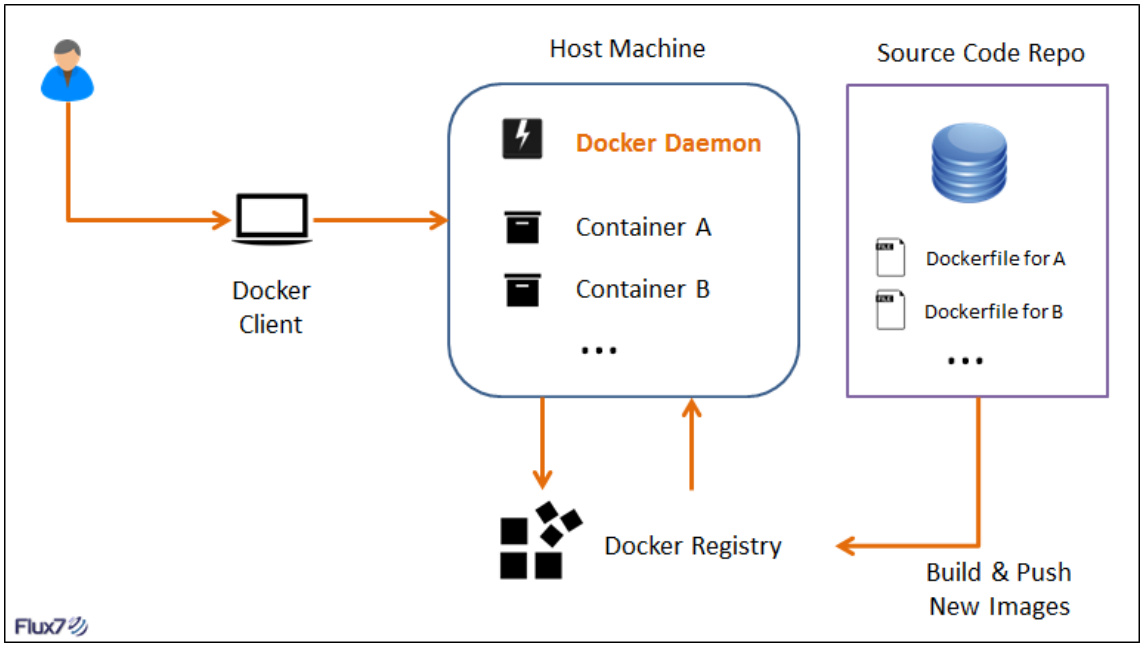
- AuFS: sistema de ficheros con una parte de sólo lectura que puede compartirse
  - Las modificaciones forman capas que se superponen
- Nueva funcionalidad:
  - construcción automática
  - control de versiones (Git)
  - compartición mediante depósitos públicos



El ecosistema Docker dispone de 3 componentes:

1. **Imágenes** (componente constructor)
   - Las imágenes docker son básicamente plantillas de solo lectura a partir de las que se instancian contenedores
2. **Depósito** (componente distribuidor)
   - Hay un depósito común para poder subir y compartir imágenes (hub.docker.com)
3. **Contenedores** (componente ejecutor)
   - Se crean a partir de las imágenes, y contienen todo lo que nuestra aplicación necesita para ejecutarse
   - Se puede convertir un contenedor en imagen mediante `docker commit`

Internamente Docker consta de un *daemon*, un depósito y una aplicación cliente (docker)



3.1 Funcionamiento

```
docker run -i -t imagen programa
```

- Acción: run = construir y ejecutar (-i -t para interactivo)
- Imagen: p.ej ubuntu
- Programa: p.ej /bin/bash

Pasos:

1. Descargar la imagen (ubuntu) desde el *Docker Hub*
2. Crear el contenedor.
3. Reservar un sistema de ficheros y añadir un nivel de lectura/escritura.
4. Reservar una interfaz de red o un bridge para comunicar con el anfitrión.
5. Reservar una dirección IP interna.
6. Ejecutar el programa especificado (/bin/bash)
7. Capturar la salida de la aplicación

3.2 Docker desde la CLI

El elemento crucial es el cliente docker mediante el que interactuamos con el demonio

```
docker [OPTIONS] COMMAND [arg...] -H, --host=[]
```

Habitualmente se precisan privilegios para su uso

Tipos de órdenes:

1. Control del ciclo de vida (run, start, stop, …)
2. Informativas (logs, ps, …)
3. Acceso al depósito (pull, push, …)
4. Otras (cp, export, …)

Imprescindibles (además de docker run):

5. docker info
6. docker pull *imagen*

7. docker ps | ps -a
8. docker images
9. docker history *imagen*

```
docker history httpd
IMAGE          CREATED        CREATED BY                                      SIZE
9a0bc463edaa   3 weeks ago    /bin/sh -c #(nop)  CMD ["httpd-foreground"]     0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  EXPOSE 80/tcp                0 B
<missing>      3 weeks ago    /bin/sh -c #(nop) COPY file:761e313354b918b6c   133 B
<missing>      3 weeks ago    /bin/sh -c set -x  && buildDeps='   bzip2   c   29.17 MB
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV HTTPD_ASC_URL=https://   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV HTTPD_BZ2_URL=https://   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV HTTPD_SHA1=5101be34ac4   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV HTTPD_VERSION=2.4.23     0 B
<missing>      3 weeks ago    /bin/sh -c apt-get update  && apt-get install   41.15 MB
<missing>      3 weeks ago    /bin/sh -c #(nop)  WORKDIR /usr/local/apache2   0 B
<missing>      3 weeks ago    /bin/sh -c mkdir -p "$HTTPD_PREFIX"  && chown   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV PATH=/usr/local/apache   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  ENV HTTPD_PREFIX=/usr/loca   0 B
<missing>      3 weeks ago    /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
missing>       3 weeks ago    /bin/sh -c #(nop) ADD file:23aa4f893e3288698c   123 MB
```

3.3 Algunos ejemplos de uso

Los apartados aplicados, como éste, únicamente pueden ser comprobados en nuestros equipos virtuales de `portal-ng`

3.3.1 Servidor de web mínimo sobre Fedora

En este apartado pretendemos tomar contacto con un despliegue artesano, realizado de manera progresiva. El servicio a desplegar, al finalizar, consistirá en un servidor de web conectado al puerto 80 que devuelve al navegador (url `http://localhost/`) una página de bienvenida:

Este primer caso se ha ideado para emplearse en el laboratorio; nos sirve para poner a prueba nuestra infraestructura: software, configuración, puertos, etc… Los pasos[3] que realizamos para lograr este objetivo requieren previamente que averigüemos qué necesitamos para construir el servicio. Abreviando, nuestra lista de requisitos es:

[3] basado en http://linoxide.com/linux-how-to/interactively-create-docker-container/

1. Una instalación base compatible con un servidor de web (p.ej. Fedora y el servidor APACHE). Es necesario prever cómo se consigue instalar el servidor sobre la imagen Fedora.
2. Material para *poblar* el directorio con documentos del servidor (p.ej. el contenido de `misitio.tar.gz`, que se encuentra en el directorio del Seminario 4 en PoliformaT[4]). Incluye carpetas con páginas, estilos, etc. Es necesario conocer dónde debe colocarse cada una de las piezas.
3. Vía de acceso al servicio resultante. Si ponemos en funcionamiento un navegador en la máquina virtual, ¿con qué url accedemos al servicio del contenedor?. Debemos indicar en el anfitrión (nuestra virtual) que uno de sus puertos (el 80) se asocie a un puerto (también el 80) del contenedor.

Vayamos por pasos, pero sin perder la perspectiva global. Conectamos con nuestra máquina virtual que dispone del software Docker necesario, y de una conexión a Internet.
Ejecutamos:

```
docker run -i -t fedora bash
```

En esta instrucción indicamos que deseamos ejecutar una imagen llamada "fedora[5]". Como Docker no dispone localmente de ninguna imagen con tal identificador, conectará con el depósito central (*docker hub*), encontrará esa imagen[6] y la traerá (204.4MBs). Cuando consiga ponerlo en marcha, ejecutará en el contenedor el shell bash y quedará a la espera de nuestras órdenes desde el teclado.
Usamos dentro del contenedor la orden `dnf` (¡ya estamos en Fedora!) que es un instalador de paquetes. Primero lo ponemos al día (`dnf update`) y después añadimos el servidor APACHE (paquete `httpd`)

```
(dentro del contenedor) dnf update
(dentro del contenedor) dnf install httpd
```

En las últimas pruebas, la primera acción necesitaba traer cerca de 94 paquetes (59MBs), y la segunda requiere 7 paquetes (4.7MBs). Una enorme ventaja de Docker es que guarda copia local de aquello que pueda reutilizar, de modo que una nueva imagen requeriría un tiempo cero.
De momento no necesitamos nada más dentro del contenedor, así que salimos de él y volvemos a nuestra virtual.

```
(dentro del contenedor) exit
```

En líneas generales, todo lo que hacemos con los contenedores es volátil salvo que digamos lo contrario.
Averiguar el nombre interno del contenedor. Como ya hemos terminado con él, buscaremos un contenedor basado en fedora que haya finalizado recientemente (unos minutos a lo sumo)

```
docker ps -a
//elegimos el más reciente basado en fedora
// copiamos los primeros dígitos de la columna CONTAINER_ID
```

[4] También en `tsr/lab3/misitio_sem4_demo` del directorio asigDSIC en los laboratorios
[5] Se trata de una distribución de LINUX emparentada con RedHat
[6] De hecho es una familia en la que cada miembro fedora tiene asociado un número de versión. Si se omite, se traerá el último

```
// y hacemos un commit
docker commit CONTAINER_ID fedora-24-httpd
```

Como resultado, si ejecutamos `docker images` observaremos dos elementos: la imagen fedora descargada del docker hub, y nuestra fedora-24-httpd que acabamos de crear localmente.

```
docker images
```

Con esto terminamos con el primer requisito y pasamos al segundo (poblar directorio). Copiamos misitio.tar.gz a nuestra virtual y lo descomprimimos, dando lugar a una carpeta misitio. Ahora hemos de instruir al contenedor para que "coja" esos materiales. Recurriremos a operaciones automatizadas a través del archivo de configuración Dockerfile[7] cuyo contenido será:

```
FROM fedora-24-httpd
ADD misitio/index.html /var/www/html/index.html
ADD misitio/css /var/www/html/css
ADD misitio/js /var/www/html/js
ADD misitio/fonts /var/www/html/fonts
EXPOSE 80
ENTRYPOINT [ "/usr/sbin/httpd" ]
CMD [ "-D", "FOREGROUND" ]
```

En ese mismo directorio ejecutamos:

```
docker build --rm -t misitio .
```

¡No olvides el punto al final de la orden! Con esto instruimos a Docker para que a partir del `Dockerfile` del directorio actual genere un contenedor llamado `misitio` eliminando los niveles intermedios[8] que pueda ir generando.

Esto finaliza el segundo requisito, y nos quedará el tercero: ejecutar el contenedor permitiendo el acceso al servicio que ofrece.

```
docker run -p 80:80 -d -P misitio
```

Esta orden pone en funcionamiento el contenedor, asociando el puerto 80 de la virtual como punto de entrada para el 80 del contenedor. Con un navegador en la máquina virtual que acceda al url `http://localhost` deberíamos ver la ilustración que aparece al comienzo de este subapartado.

Para terminar con el contenedor, averiguamos su identificador con `docker ps`[9], y lo detenemos con `docker stop`. La prueba de fuego consiste en recargar el navegador, que fallará al no poder conectar.

[7] Dockerfile es objeto de estudio en uno de los próximos apartados
[8] Cada nivel se obtiene como resultado de ejecutar una orden del Dockerfile
[9] No necesita opciones porque el contenedor se encuentra en ejecución

Conclusiones destacables

- La creación de una imagen nueva puede consumir tiempo y recursos de forma apreciable. Si la imagen se basa en otra de la que ya disponemos, el sistema reduce los pasos necesarios y el consumo de recursos. Ésta es una de las razones que aconsejan reducir razonablemente la variedad de imágenes.
- Un Dockerfile completo permite reproducir las acciones con comodidad y reduciendo el riesgo de errores, pero es difícil escribirlo correctamente a la primera. El método de prueba y error mediante una sesión interactiva es una buena aproximación. Consultar la documentación es **imprescindible**.

3.3.2 Preparar contenedor para ejecutar aplicaciones NodeJS con ZeroMQ

1. Lanzar Docker utilizando una imagen Fedora interactiva:

```
docker run –i –t fedora bash
```

2. Desde el intérprete de órdenes de ese contenedor, iremos lanzando las siguientes órdenes:

```
dnf install –y nodejs
dnf install –y make
dnf install –y zeromq-devel
dnf install –y python gcc-c++
npm install zmq
```

3. Obtener el identificador o nombre del contenedor utilizado en los pasos anteriores, desde la línea de órdenes de nuestro sistema (anfitrión):

```
docker ps
CONTAINER ID   IMAGE    COMMAND   CREATED       ...  NAME
c7b87a37d012   fedora   “bash”    9 minutes ago      serene_hopper
```

4. Realizar el `commit` del contenido actual del contenedor, generando así una nueva imagen

```
docker commit serene_hopper zmq
```

o, alternativamente

```
docker commit c7b87 zmq
```

   - Ahora ya tendremos una imagen Docker llamada “zmq” desde la que lanzar programas node en contenedores.

5. Comprobar mediante `docker images`

3.4 Pequeños trucos

1. **Identificadores más recientes**

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit $(dl) helloworld
```

2. **Consultar IP**

```
docker inspect $(dl) | grep IPAddress | cut -d '"' -f 4
```

3. **Asignar puerto**

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf 0).HostPort}} {{end}}' <id_contenedor>
```

4. **Localizar contenedores mediante expresión regular**

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done
```

5. **Consultar entorno**

```
docker run --rm ubuntu env
```

6. **Finalizar todos los contenedores en ejecución**

```
docker kill $(docker ps -q)
```

7. **Eliminar contenedores viejos**

```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

8. **Eliminar contenedores detenidos**

```
docker rm -v $(docker ps -a -q -f status=exited)
```

9. **Eliminar imágenes *colgadas***

```
docker rmi $(docker images -q -f dangling=true)
```

10. **Eliminar todas las imágenes**

```
docker rmi $(docker images -q)
```

11. **Eliminar volúmenes *colgadas***

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

(En 1.9.0, dangling=false no funciona y muestra todos los volúmenes)

12. **Mostrar dependencias entre imágenes**

```
docker images -viz | dot -Tpng -o docker.png
```

13. **Monitorizar el consumo de recursos al ejecutar contenedores**

Para averiguar el consumo de CPU, memoria y red de un único contenedor, puedes usar:

```
docker stats <container>
```

Para todos los contenedores, ordenados por id:

```
docker stats $(docker ps -q)
```

Ídem ordenados por nombre:

```
docker stats $(docker ps --format '{{.Names}}')
```

Ídem seleccionando los que proceden de una imagen:

```
docker ps -a -f ancestor=ubuntu
```

4 EL FICHERO DE PROPIEDADES DOCKERFILE

Docker puede **construir una imagen** a partir de las instrucciones de un fichero de texto llamado Dockerfile, que debe encontrarse en la raíz del depósito que deseemos construir

```
docker build <ruta_depósito> <opciones>
```

Sintaxis general: INSTRUCCIÓN argumento

- Por convención, las instrucciones se escriben en mayúsculas
- Se ejecutan por orden de aparición en el Dockerfile

Todo Dockerfile debe comenzar con **FROM**, que especifica la imagen que se toma como base para construir la nueva.

- Sintaxis: FROM <nombre_imagen>

4.1 Órdenes en Dockerfile

Dispones de una referencia completa, en inglés, en la "Documentación adicional sobre Docker"

1. MAINTAINER: Establece el autor de la imagen
   - Sintaxis: MAINTAINER <nombre autor>
2. RUN: Ejecuta una orden (shell o exec), añadiendo un nuevo nivel sobre la imagen resultante. El resultado se toma como base para la siguiente instrucción
   - Sintaxis: RUN <orden>
3. ADD: Copia archivos de un lugar a otro
   - Sintaxis: ADD <origen> <destino>
   - El origen puede ser un URL, un directorio (se copia todo su contenido) o un archivo accesible en el contexto de esta ejecución
   - El destino es una ruta en el contenedor
4. CMD: Esta orden proporciona los valores por defecto en la ejecución del contenedor
   Sólo puede usarse una vez (si hubiera varias, sólo se ejecutará la última)
   - Sintaxis: 3 alternativas
     - CMD ["ejecutable","param1","param2"]
     - CMD ["param1","param2"]
     - CMD orden param1 param2
5. EXPOSE: Indica el puerto en el que el contenedor atenderá (*listen*) peticiones
   - Sintaxis: EXPOSE <puerto>
6. ENTRYPOINT: Configura un contenedor como si fuera un ejecutable
   - Especifica una aplicación que se ejecutará automáticamente cada vez que se instancie un contenedor a partir de esta imagen
     - Implica que éste será el único propósito de la imagen
   - Como en CMD, sólo se ejecutará el último ENTRYPOINT especificado
   - Sintaxis: 2 alternativas
     - ENTRYPOINT ['ejecutable', 'param1','param2']
     - ENTRYPOINT orden param1 param2
7. WORKDIR: Establece el directorio de trabajo para las instrucciones RUN, CMD y ENTRYPOINT.
   - Sintaxis: WORKDIR /ruta/a/directorio_de_trabajo
8. ENV: Asigna valores a las variables de entorno que pueden ser consultadas por los programas dentro del contenedor.

- Sintaxis: ENV <variable> <valor>

9. USER: Establece el UID bajo el que se ejecutará la imagen.
   - Sintaxis: USER <uid>
10. VOLUME: Permite el acceso del contenedor a un directorio del anfitrión.
   - Sintaxis: VOLUME ['/datos']

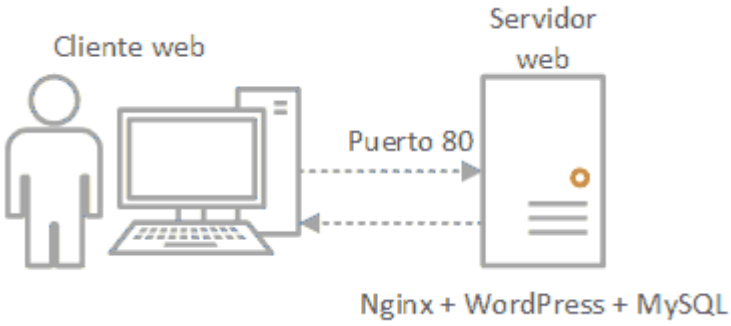
4.2 Ejemplos de Dockerfile

Puedes consultar otros ejemplos en la documentación adicional y en muchos lugares de la web, como https://github.com/komljen/dockerfile-examples

4.2.1 Ejemplo 1 (WordPress sobre Ubuntu)

Este primer ejemplo trata de un servicio web basado en WordPress instalado sobre Ubuntu (https://github.com/eugeneware/docker-wordpress-nginx/blob/master/Dockerfile). Requiere:

- Un servidor web (Nginx en este caso)
- Soporte para el lenguaje PHP
- Un servidor de BBDD (MySQL típicamente)



Ideado para un único componente

```
FROM ubuntu:14.04
MAINTAINER Eugene Ware <eugene@noblesamurai.com>
# Keep upstart from complaining
RUN dpkg-divert --local --rename --add /sbin/initctl
RUN ln -sf /bin/true /sbin/initctl
# Let the container know that there is no tty
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get -y upgrade
# Basic Requirements
RUN apt-get -y install mysql-server mysql-client nginx php5-fpm php5-mysql php-apc
pwgen python-setuptools curl git unzip
# Wordpress Requirements
RUN apt-get -y install php5-curl php5-gd php5-intl php-pear php5-imagick php5-imap
php5-mcrypt php5-memcache php5-ming php5-ps php5-pspell php5-recode php5-sqlite php5-
tidy php5-xmlrpc php5-xsl
RUN sed -i -e"s/^bind-address\s*=\s*127.0.0.1/bind-address = 0.0.0.0/"
/etc/mysql/my.cnf
# nginx config
RUN sed -i -e "s/keepalive_timeout\s*65/keepalive_timeout 2/" /etc/nginx/nginx.conf
RUN sed -i -e "s/keepalive_timeout 2/keepalive_timeout 2;\n\tclient_max_body_size
100m/" /etc/nginx/nginx.conf
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
# php-fpm config
RUN sed -i -e "s/;cgi.fix_pathinfo=1/cgi.fix_pathinfo=0/g" /etc/php5/fpm/php.ini
RUN sed -i -e "s/upload_max_filesize\s*=\s*2M/upload_max_filesize = 100M/g"
/etc/php5/fpm/php.ini
RUN sed -i -e "s/post_max_size\s*=\s*8M/post_max_size = 100M/g" /etc/php5/fpm/php.ini
RUN sed -i -e "s/;daemonize\s*=\s*yes/daemonize = no/g" /etc/php5/fpm/php-fpm.conf
RUN sed -i -e "s/;catch_workers_output\s*=\s*yes/catch_workers_output = yes/g"
/etc/php5/fpm/pool.d/www.conf
```

```
RUN find /etc/php5/cli/conf.d/ -name "*.ini" -exec sed -i -re 's/^(\s*)#(.*)/\1;\2/g'
{} \;
# nginx site conf
ADD ./nginx-site.conf /etc/nginx/sites-available/default
# Supervisor Config
RUN /usr/bin/easy_install supervisor
RUN /usr/bin/easy_install supervisor-stdout
ADD ./supervisord.conf /etc/supervisord.conf
# Install Wordpress
ADD http://wordpress.org/latest.tar.gz /usr/share/nginx/latest.tar.gz
RUN cd /usr/share/nginx/ && tar xvf latest.tar.gz && rm latest.tar.gz
RUN mv /usr/share/nginx/html/5* /usr/share/nginx/wordpress
RUN rm -rf /usr/share/nginx/www
RUN mv /usr/share/nginx/wordpress /usr/share/nginx/www
RUN chown -R www-data:www-data /usr/share/nginx/www
# Wordpress Initialization and Startup Script
ADD ./start.sh /start.sh
RUN chmod 755 /start.sh
# private expose
EXPOSE 3306
EXPOSE 80
CMD ["/bin/bash", "/start.sh"]
```

4.2.2 Ejemplo 2 (NodeJS y ZMQ)

Las acciones del anterior ejemplo 3 (apartado 3.3.2) podrían recogerse en este Dockerfile:

```
#Take the latest Fedora distribution as a base.
# Currently (November 9, 2015), it is Fedora 23.
FROM fedora
# Install the latest NodeJS on that distribution.  It is NodeJS v6.6.0.
# We may use the dnf Fedora command to this end.
RUN dnf install -y nodejs
# Next step: Install the zeromq library. Its package is called "zeromq-devel" in the
Fedora
# distribution.
RUN dnf install -y zeromq-devel
RUN dnf install -y python gcc-c++
# We also need the "npm" node module installer in order to complete our work. So, let
us
# proceed to install it.
# Warning! This is a long step, since it needs to install 115 Fedora packages!!
RUN dnf install -y npm
# We still need the standard "make" command in order to run "npm".
RUN dnf install -y make
# Finally, let us run the npm command for installing the "zmq" module.
RUN npm install zmq
```

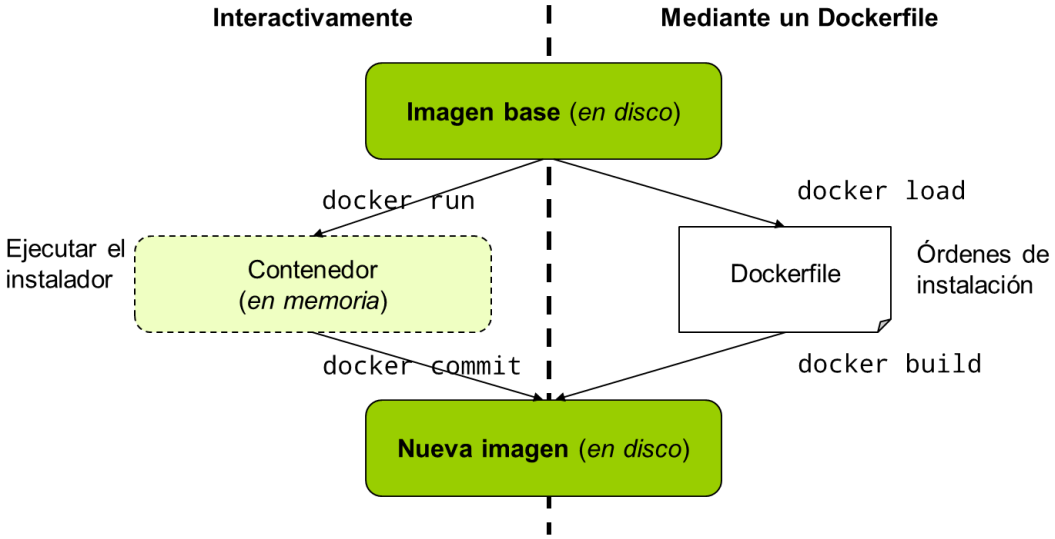
La imagen “zmq” generada entonces podría construirse ahora mediante...

```
docker build –t zmq .
```

- La opción “-t” permite asignar una etiqueta o nombre a la imagen creada.
- Esta orden debe ser lanzada en el directorio donde resida el Dockerfile visto en la hoja anterior.

4.3 Construcción de una imagen Docker como proceso

Tal y como hemos visto, podemos proceder a construir una imagen de forma progresiva e interactiva (*prueba y error*) mediante algún shell sobre el contenedor; iríamos anotando los pasos que nos parezcan apropiados y, de esta forma, construiríamos el Dockerfile que automatice la generación de la imagen.



Aunque el número de imágenes disponibles, producidas por otra gente, es muy elevado, no suele encontrarse ninguna que se ajuste exactamente a nuestras necesidades. Sin embargo sí que es muy habitual que varias de ellas puedan ser tomadas como referencia para añadir posteriormente nuestro middleware y nuestra aplicación final.

5 MÚLTIPLES COMPONENTES

El anterior ejemplo de WordPress mezcla en un mismo contenedor varios componentes, perdiendo oportunidades de escalado y disponibilidad

- Una mejora mínima consiste en diferenciar el SGBD (MySQL) y ubicarlo en otro contenedor

El despliegue ahora se complica porque se encuentran dependencias que deben ser resueltas:

- ¿cómo sabe el primer componente (Nginx+WordPress) los detalles necesarios para conectar con el segundo (MySQL)?
  - Al menos su IP (resto de valores por defecto)
- Puede que no se resuelvan hasta que el segundo componente inicie su ejecución

Simplificadamente, se requiere …

1. Crear los Dockerfile de ambos componentes
   - Serán un subconjunto del mostrado anteriormente
2. Iniciar el segundo, obteniendo su IP
3. Iniciar el primero, transmitiéndole la IP del segundo

Sin embargo, esta aproximación artesana es inaplicable para casos de envergadura media y grande. Se necesita:

- Lenguaje para generalizar la descripción de los despliegues
  - Que pueda expresar componentes, propiedades y relaciones
  - P.ej. Compose-YAML, OASIS-TOSCA
- Automatizar la ejecución de los despliegues
  - Mediante un motor que ejecuta el despliegue según la descripción
  - P.ej. Docker-Compose, APACHE-Brooklyn

Es conveniente disponer de herramientas que faciliten la creación y simulación de estos despliegues, como https://lorry.io/

- ¡¡Los imprevistos no son bienvenidos mientras se realiza un despliegue de gran magnitud!!

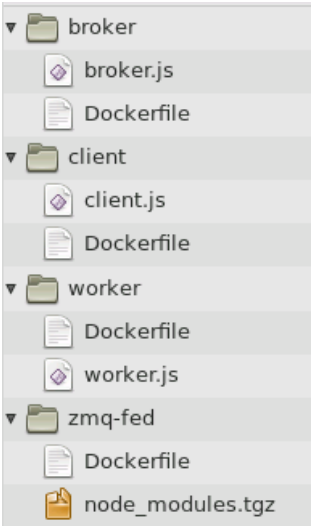
Docker admite “enlaces” entre contenedores cuyo establecimiento puede **automatizarse mediante “docker-compose**”.

5.1 Ejemplo: client/ broker/worker con ZMQ

Ejemplo: Servicio implantado mediante un componente "client", otro "broker" (tipo ROUTER-DEALER) y un "worker", que podrá replicarse tantas veces como sea necesario. Tanto el cliente como el trabajador necesitan como argumento el URL correspondiente del broker.

5.1.1 client.js

Requiere el URL del broker como argumento




```
// client.js
// Client process. It sends a random integer value to
// the servers, and it prints the returned result.

// Import the zmq module.
var zmq = require('zmq');
// Count the number of replies.
var numReplies = 0;
// Total number of requests.
const reqs = 10;

// Check whether we have received the URL as an argument.
if (process.argv.length != 3) {
  console.error("The broker URL should be given as an
argument!!");
  process.exit(1);
}
// Get the broker URL.
var url = process.argv[2];
console.log("Broker URL is %s.", url);
var req = zmq.socket('req');
// Connect to the broker.
req.connect(url);

// Handle the received replies.
req.on('message', function(msg) {
       // Get the reply value.
       var value = parseInt(msg);
       // Print the value.
       console.log("Returned value: %d", value);
       // Check whether we should terminate.
       if (++numReplies == reqs)
              process.exit(3);
});

// Send a sequence of 10 requests
for (var i = 0; i < reqs; i++) {
       // Generate a random integer value in the range 1..100
       var value = Math.floor(Math.random()*100 + 1);
       // Print the request.
       console.log("Request: %d", value);
       // Send it.
       req.send(value.toString());
}
```

5.1.2 broker.js

```
// broker.js
// Basic ROUTER-DEALER broker.

var zmq = require('zmq');
var router = zmq.socket('router');
var dealer = zmq.socket('dealer');

// Bind these sockets to their ports.
router.bindSync("tcp://*:8000");
dealer.bindSync("tcp://*:8001");

// The router simply forwards messages
// in both directions...
router.on('message', function(){
  var args = Array.apply(null, arguments);
  dealer.send(args);
});

dealer.on('message', function() {
  var args = Array.apply(null, arguments);
  router.send(args);
});
```

5.1.3 worker.js

Requiere el URL del broker como argumento

```
// worker.js
// It assumes that requests only carry an integer value.
// It returns the double of the incoming value.

// Import the zmq module.
var zmq = require('zmq');

// Check whether we have received the URL as an argument.
if (process.argv.length != 3) {
  console.error("The broker URL should be given as an argument!!");
  process.exit(1);
}
// Get the broker URL.
var url = process.argv[2];
console.log("Worker: The broker URL is %s", url);

// Create the REP socket.
var rep = zmq.socket('rep');
// Connect to the broker.
rep.connect(url);

// Handle the received requests.
rep.on('message', function(msg) {
  // Get the incoming value.
  var value = parseInt(msg);
  // Double it!
  value *= 2;
  // Print the reply.
  console.log("Reply: %d", value);
  // Send the reply.
  rep.send(value.toString());
```

```
});
```

Nos basamos en una imagen Docker llamada "zmq-fed" con...

- NodeJS instalado sobre una imagen de alguna distribución[10] Linux.
- La biblioteca zeromq instalada correctamente.
- El módulo zmq instalado y disponible para su uso desde node.
- Sobre esta imagen se podrá ejecutar cualquier programa NodeJS que utilice ØMQ.

El Dockerfile para crear dicha imagen "**zmq-fed**[11]" es:

```
# Take the latest Fedora distribution as a base.
# Currently (November 9, 2016), it is Fedora 24.
FROM fedora
# Install the latest Node.js on that distribution.
# This also installs npm.
RUN curl -sL https://rpm.nodesource.com/setup_6.x | bash -
RUN dnf install -y nodejs
# Next step: Install the zeromq library.
# Its package is called "zeromq-devel" in the Fedora
# distribution.
RUN dnf install -y zeromq-devel
# The following lines install the "node_modules" directory
# generated with a "npm install zmq" order on a system with
# the required tools. Those tools are not installed in the
# image being built here.
ADD ./node_modules.tgz /
```

Pasos a seguir para utilizar este servicio...

1. Crear la imagen Docker para el componente **broker**.

Necesitamos un Dockerfile similar a éste:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./broker.js /zmq/broker.js
WORKDIR /zmq
EXPOSE 8000 8001
CMD node broker.js
```

- Se deja público el puerto 8000 para los clientes...
- ... y el 8001 para los "*workers*".

Y generamos su contenedor con esta orden:

```
docker build –t broker .
```

- Lanzamos el broker (**docker run**) y averiguamos su IP mediante **docker inspect broker**

[10] Fedora en este caso

[11] Este Dockerfile y los restantes de esta sección, así como el fichero node_modules.tgz referenciado en los Dockerfile deben bajarse desde PoliformaT. Se encuentran en un ZIP llamado `dcExample.zip` en la carpeta de recursos correspondiente a este Seminario 4.

- Anotamos la IP que aparece (p.ej. *a.b.c.d*) y la usamos para construir el Dockerfile de los otros 2 componentes

2. Crear la imagen Docker para el componente **client**.

Necesitamos un Dockerfile similar a éste:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./client.js /zmq/client.js
WORKDIR /zmq
# We assume that each worker is linked to the broker
# container.
CMD node client tcp://a.b.c.d:8000
```

- Y generamos su imagen desde el directorio client con esta orden Docker:

```
docker build -t client .
```

3. Crear la imagen Docker para el componente **worker**.

Necesitamos un Dockerfile similar a éste:

```
FROM zmq-fed
RUN mkdir /zmq
COPY ./worker.js /zmq/worker.js
WORKDIR /zmq
# We assume that each worker is linked to the broker
# container.
CMD node worker tcp://a.b.c.d:8001
```

- Y generamos su imagen con esta orden estando en el directorio worker:

```
docker build -t worker .
```

Ahora ya podemos ejecutar estos procesos comprobando que conectan con el broker y que envían y reciben mensajes.

6 DESPLIEGUE EN DOCKER COMPOSE

URL https://docs.docker.com/compose/

Docker Compose es una herramienta para definir y ejecutar aplicaciones ubicadas en varios contenedores Docker.

Tres pasos:

1. Definir el entorno de la aplicación con un **Dockerfile**
2. Definir los servicios que constituyen la aplicación en un archivo **docker-compose.yml** para que puedan ejecutarse conjuntamente
3. Ejecutar **docker-compose up**, con lo que Compose iniciará y ejecutará la aplicación completa

Limitado a contenedores en el mismo equipo, pero puede completarse con Docker Swarm (u otro software de orquestación) para controlar un cluster.

Es una aplicación que se instala aparte de Docker.

6.1 Características destacables

- Se puede disponer de varios entornos aislados en un mismo equipo
- Se pueden ubicar los datos fuera de los contenedores (volúmenes)
- Sólo hay que crear de nuevo los contenedores que se hayan modificado
- En docker-compose.yml se pueden usar capacidades para adaptarse al entorno

El elemento crucial es el programa docker-compose

```
docker-compose [OPTIONS] [COMMAND] [ARGS...]
```

Ciclo típico de uso:

```
$ docker-compose up -d
… actividades …
$ docker-compose stop
$ docker-compose rm –f
```

6.2 Docker Compose desde la CLI

Órdenes más significativas:

- **build**: (re-)construye un servicio
- **kill**: detiene un contenedor
- **logs**: muestra la salida de los contenedores
- **port**: muestra el puerto asociado al servicio
- **ps**: lista los contenedores
- **pull**: sube una imagen
- **rm**: elimina un contenedor detenido
- **run**: ejecuta una orden en un servicio
- **scale**: número de contenedores a ejecutar para un servicio
- **start** | **stop** | **restart**: inicia | detiene | reinicia un servicio
- **up**: build + start (*aproximadamente*)

Ejemplo de orden **run**

```
$ docker-compose run web python manage.py shell
```

1. Inicia el servicio **web**

2. Envía al servicio la orden **python manage.py shell** para que la ejecute
3. La ejecución de esa orden en el servicio se desvincula de nuestro shell (desde donde lanzamos docker-compose)

Hay dos diferencias entre **run** y **start**

- La orden que pasamos a run tiene preferencia sobre la que se haya especificado en el contenedor
- Si hay puertos en colisión con otros ya abiertos, no se crearán otros

6.3 El fichero de descripción del despliegue docker-compose.yml

https://docs.docker.com/compose/compose-file/

Es un archivo que sigue la sintaxis YAML

Además de las órdenes análogas a los parámetros de "docker run", las principales son:

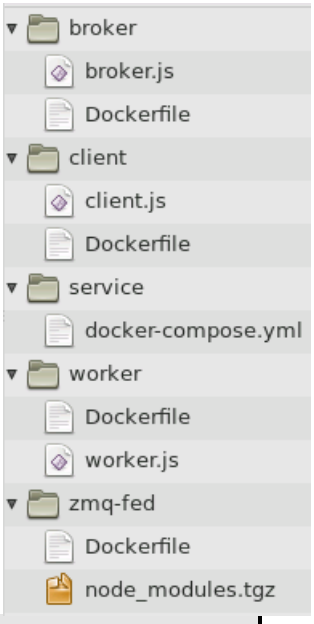
- **image**: referencia local o remota a una imagen, por nombre o tag
- **build**: ruta a un directorio que contiene un Dockerfile
- **command**: cambia la orden a ejecutar en el inicio
- **links**: enlace a contenedores de otro servicio.
- **external_links**: enlaces a contenedores externos a compose
- **ports**: puertos expuestos (mejor expresarlos entre comillas)
- **expose**: Ídem, pero accesible sólo a servicios enlazados (con links)
- **volumes**: monta rutas como volúmenes

6.4 Revisitando un ejemplo anterior

En el ejemplo con `client`, `broker` y `worker` desplegado anteriormente (apartado 5.1), se ha resuelto artesanalmente la dependencia del resto de componentes respecto al `broker`: necesitaban conocer su IP. La forma en que se ha resuelto es incompatible con la automatización del despliegue, incluso con el momento en que cada información se requiere: ¡necesitamos ejecutar el broker antes de poder desplegar el resto!

- ¿Si se cambiara la IP del `broker` habría que re-desplegar de nuevo `client` y `worker`?

Debe haber una solución mejor, y ésa es una de las aportaciones de `docker-compose` y su fichero de descripción del despliegue. Crearemos una carpeta `service` para él.



Supongamos que el `broker` pueda anunciar detalles sobre sí mismo (como su IP), y que existe una forma de inyectar esa información en los componentes adecuados. Esto lo podemos especificar en un archivo de configuración del despliegue `docker-compose.yml`:

```
version: '2'
services:
  wor:
    image: worker
    build: ../worker/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:8001
```

```
  cli:
    image: client
    build: ../client/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:8000
  bro:
    image: broker
    build: ../broker/
```

También necesitamos que los `Dockerfile` de `client` y `worker` puedan consultar esa información. Observaréis que hemos colocado el URL completo para acceder al socket ZMQ del `broker`, por lo que la última línea del `Dockerfile` de los componentes debería cambiarse por la siguiente:

- En el `Dockerfile` del cliente:

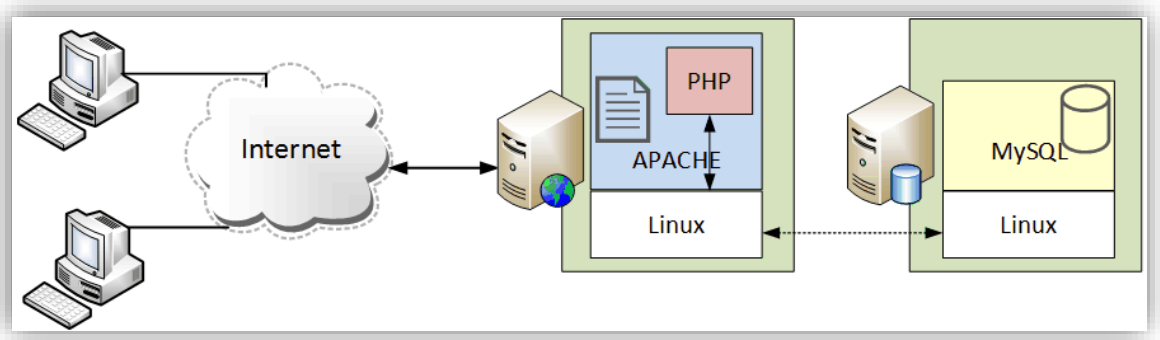
```
CMD node client $BROKER_URL
```

- En el `Dockerfile` del trabajador:

```
CMD node worker $BROKER_URL
```

6.5 Ejemplo: WordPress sobre Fedora

El segundo caso que ponemos en funcionamiento sólo pretende ilustrar cómo se puede poner en marcha un servicio completo y complejo con facilidad. En el tema 1 ya se describió la Wikipedia como un sistema LAMP. El ejemplo que abordamos a continuación, **WordPress**, está muy relacionado con esa misma tecnología (LAMP). La idea, resumida, es abordar cómo instalarlo y ponerlo en marcha con unas cuantas instrucciones, para después discutir y modificar sus características.



Como en anteriores ocasiones, la secuencia de operaciones a realizar supone el uso de un entorno como el de nuestro laboratorio de máquinas virtuales. Para probarlo allí se debe crear un directorio wordpress y realizar en él las actividades[12] siguientes.

1. (*) Descargamos la última versión de WordPress:

[12] Descargando `tsr_lab3_material.zip` de PoliformaT podemos ahorrarnos las acciones marcadas con (*)

```
curl https://wordpress.org/latest.tar.gz | tar -xvzf -
```

2. (*) Creamos el Dockerfile con estas 2 líneas:

```
FROM orchardup/php5
ADD . /code
```

La imagen base (orchardup/php5) que se cita se basa en una distribución Ubuntu, que **costará tiempo** en descargar. Es interesante simultanear esa tarea con alguna otra para no quedarnos con los brazos cruzados.

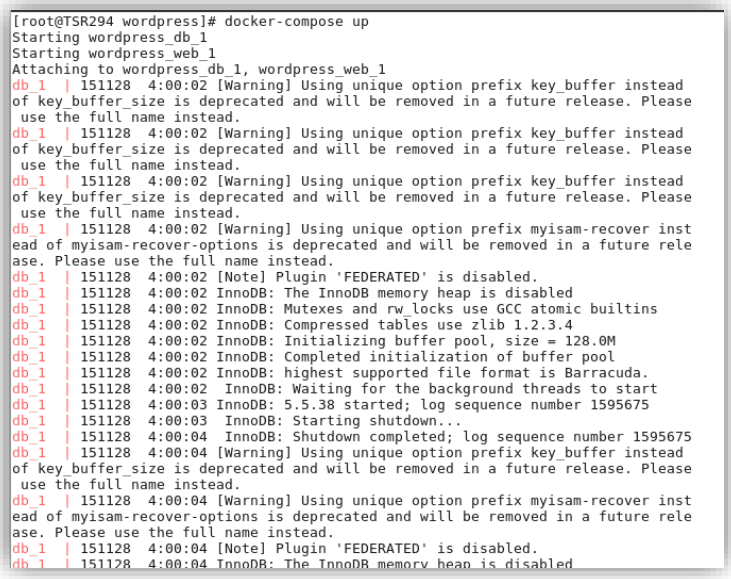
3. (*) Creamos el archivo de configuración de despliegue **docker-compose.yml**

```
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    - "8000:8000"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

El anexo sobre **docker-compose.yml** contiene información sobre su sintaxis y significado

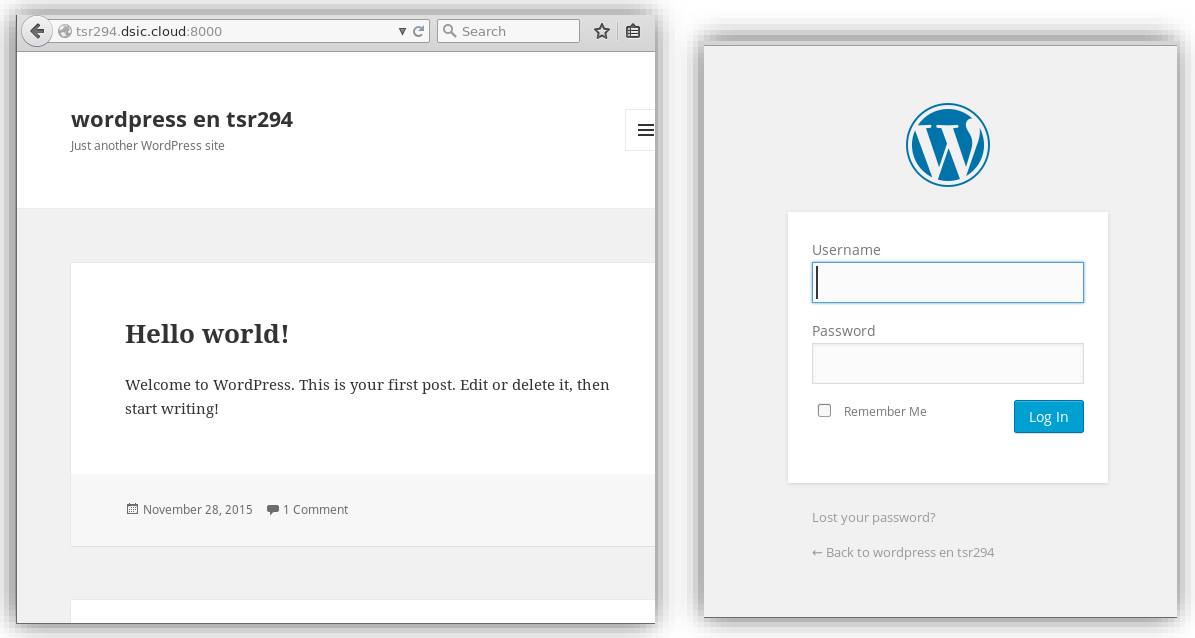
4. (*) Hay varios archivos de configuración (`wp-config.php`, `router.php`) de WordPress que se pueden/deben modificar, pero carecen de un interés específico en esta asignatura, y puedes utilizar los que vienen incluidos en **tsr_lab3_material.zip**.
5. ¡A rodar!!!...con `docker-compose up`

- ◁¡¡Recuerda detener el servicio al final!

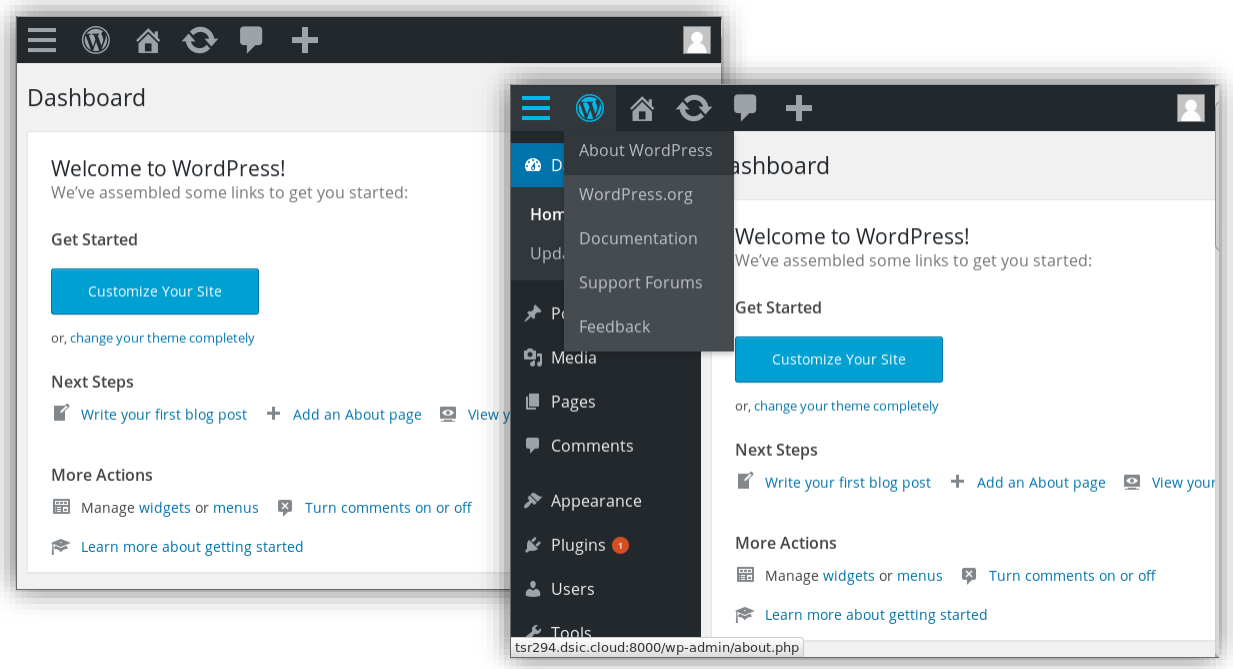


DEBES acceder con un navegador. Cuidado con los detalles, como el puerto de conexión.

El proceso de instalación de WordPress incluye una última etapa que se realiza mediante el navegador. Ésa será la página que se muestre en el primer momento.



La configuración crea una cuenta administrativa (suele denominarse admin) a la que se puede acceder como `/admin` al final del URL.



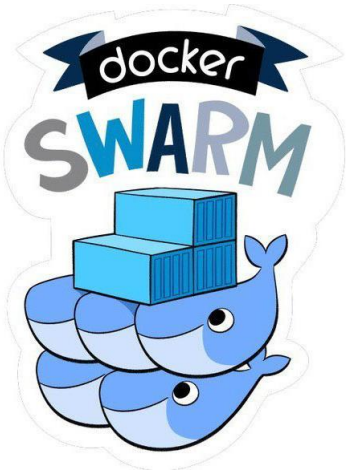
7 MÚLTIPLES NODOS

El objetivo de diseño de Compose se limita a componentes que has de ejecutar en un único nodo; sin embargo la escalabilidad no puede proceder del reparto de los recursos de un nodo entre los componentes de la aplicación, sino de la agregación de otros nodos con sus recursos a nuestra aplicación.

- La tolerancia a fallos, con un solo nodo, queda completamente desnaturalizada.
- La mera concepción de una aplicación distribuida limitada a un nodo es contradictoria.

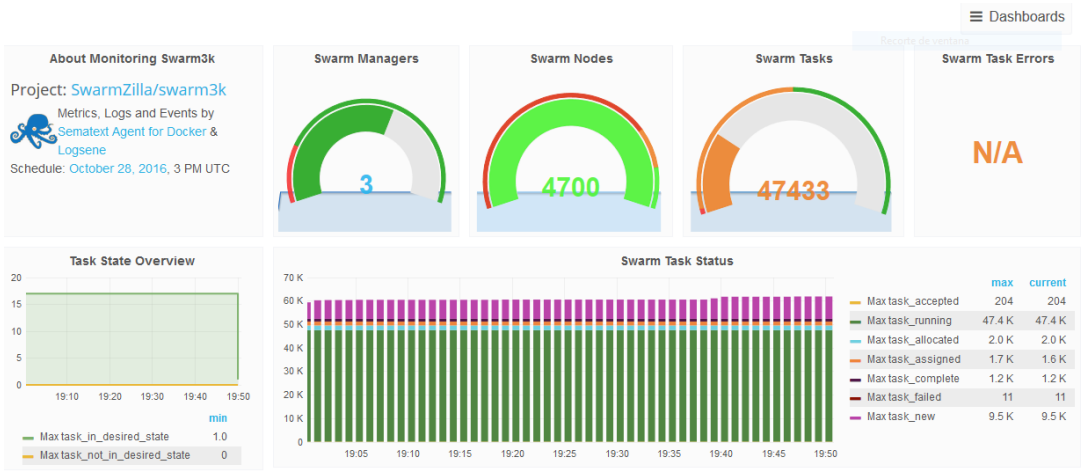
¿Qué se podría necesitar para que los sistemas Docker de múltiples nodos puedan interactuar e integrarse como si de un sistema único se tratara? Un director de orquesta que los coordine.

- El software de **orquestación** pondrá en contacto a todos los nodos entre sí, ofreciendo propiedades al sistema y funcionalidad a las aplicaciones.



En general, las aplicaciones más veteranas diseñadas con este propósito nacieron en el entorno de la virtualización y de la nube. Algunos casos destacables son Kubernetes (Google) y Apache Mesos. Con la llegada de las tecnologías de contenerización, estos sistemas también se han adaptado para interactuar con Docker, pero tienen que rivalizar con la propuesta nativa denominada "Docker en modo **Swarm**", incorporada a partir de la versión 1.12[13] de Docker.

- El experimento de escalabilidad más relevante, en Octubre de 2016 (**swarm3k**, https://github.com/swarmzilla/swarm3k), incluyó 4700 nodos de voluntarios de todo el mundo.



Pese a que nuestro tiempo de laboratorio es limitado y excluye el uso de esta opción multinodo, creemos imprescindible conocer los elementos más relevantes de Swarm.

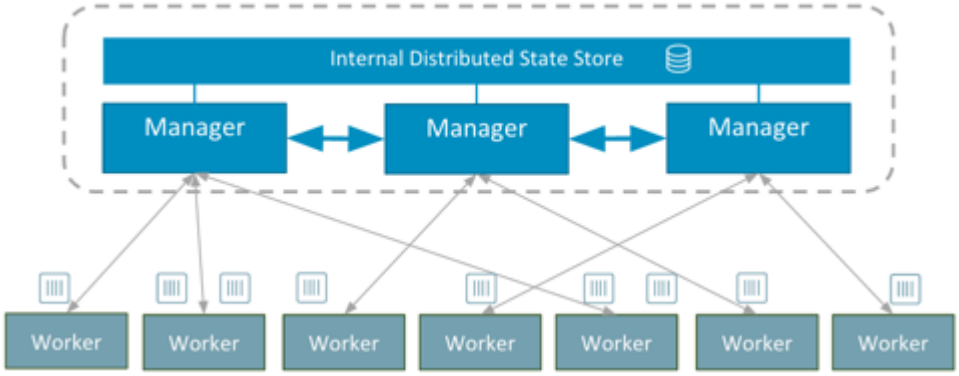
[13] Anteriormente existía un Swarm-kit limitado e incompatible con el sistema actual

7.1 Conceptos de Docker Swarm

Un **enjambre** (*swarm*) es un cluster de nodos con instancias de Docker (*Docker Engines*) sobre el que se despliegan servicios.

Un **nodo** es una instancia de Docker Engine que participa en el enjambre.

7.1.1 Tipos de nodo



1. **Manager**. Los manager se encargan de enviar las tareas a los workers, realizar operaciones para mantener el estado deseado, y elegir al líder para las tareas de orquestación.
2. **Worker**. Reciben y ejecutan las tareas encomendadas por los managers. Los nodos manager también pueden ser worker. El manager recibe información sobre el estado actual de las tareas asignadas.

7.1.2 Servicios y tareas

La **tarea** es la unidad mínima de planificación, y es asignada por el manager entre los nodos worker. Una vez asignada a un nodo, no puede migrar a otro.

El **servicio** es la definición de las tareas que deben ejecutarse en los nodos worker.

- Cuando se crea un servicio se especifica qué imagen usar y qué órdenes ejecutar dentro de los contenedores.

En el modelo de **servicios replicados**, el manager reparte cierta cantidad de tareas replicadas entre los nodos, dependiendo del escalado que se desee en el estado marcado como objetivo.

Para **servicios globales**, en cada nodo disponible del cluster el enjambre ejecuta una tarea del servicio.

7.2 Características de Docker Swarm

1. **Integrado** con Docker, sin requerir software adicional.
2. Diseño **descentralizado**. La distinción entre rol trabajador y manager se establece en tiempo de ejecución, no en el despliegue.
3. **Modelo declarativo** del servicio. La descripción de los estados aceptables de los servicios de la aplicación es suficiente para que el sistema agregue los detalles sobre cómo conseguirlo.

4. **Escalado**. Para cada servicio se puede indicar el número de tareas deseadas. Al escalar, el manager realiza automáticamente la adaptación agregando o eliminando tareas de acuerdo con el estado objetivo.
5. **Reconciliación al estado objetivo**. El nodo manager monitoriza el estado del cluster y reconcilia cualquier diferencia existente respecto al estado marcado como objetivo.
    - P.ej. si se ha establecido que deben mantenerse 10 réplicas de un contenedor y un nodo con dos de esas instancias falla, el manager creará otras dos para reemplazarlas en nodos que se encuentren disponibles.
6. **Redes de solapamiento**[14]. Superponiéndose a las redes reales, se puede especificar una red solapada en la que cada contenedor recibe una dirección.
7. **Descubrimiento de servicios**. Los manager asignan a cada servicio un nombre DNS único, y distribuye el servicio equilibrando la carga entre los contenedores.
8. **Equilibrado de carga**. Se puede colocar un repartidor de carga externos entre los puertos expuestos para los servicios. Swarm permite repartir los contenedores de servicio entre los nodos.
9. **Seguridad por defecto**. Cada nodo se comunica con el resto mediante TLS, con la opción de establecer nuestra propia autoridad de certificación.
10. **Actualizaciones reversibles**[15]. Las actualizaciones de servicios aplicadas a los nodos son controladas por el manager. En caso de error, se puede revertir la actualización.

7.3 Pasos típicos en el uso de Swarm

En el tutorial de Swarm (https://docs.docker.com/engine/swarm/swarm-tutorial/) puedes encontrar el detalle acerca del uso de este orquestador para un servicio mínimo, incluyendo:

1. Inicializar un cluster en modo swarm
2. Agregar nodos al enjambre
3. Desplegar servicios en el enjambre
4. Gestionar el enjambre cuando ya se encuentra todo en funcionamiento

8 BIBLIOGRAFÍA

- www.docker.com (lugar oficial de Docker) Atención a las diferencias entre las versiones 1 y 2 de Compose, y a la posible confusión entre SwarmKit (obsoleto) y modo Swarm
    - docs.docker.com/userguide/ (**documentación oficial**)
    - docs.docker.com/compose/ (Compose)
        - docker-compose.yml: https://docs.docker.com/compose/compose-file/
    - docs.docker.com/engine/swarm/ (modo Swarm)
        - Esquemas con detalles sobre su funcionamiento interno en http://events.linuxfoundation.org/sites/events/files/slides/ContainerCon%20NA%20%281%29.pdf
- www.mindmeister.com/389671722/docker-ecosystem

[14] *overlay*
[15] *rolling*

    - Mapa conceptual (¡enorme!) sobre Docker/Open Container
- flux7.com/blogs/docker/ (blog sobre Docker)
- Consejos sobre Dockerfile: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
- Recopilatorio de información sobre Docker: https://github.com/wsargent/docker-cheat-sheet
- 12factor.net/ (metodología de diseño de aplicaciones "*The twelve-factor app*")
- Póster sobre el ecosistema Docker: http://comp.photo777.org/wp-content/uploads/2014/09/Docker-ecosystem-8.6.1.pdf
- Lista exhaustiva de enlaces sobre Docker: https://github.com/veggiemonk/awesome-docker

9 APÉNDICE. EJEMPLOS DOCKERFILE

En este apartado damos cabida a algunos ejemplos que pretenden ilustrar desde el uso de Docker para recubrir el despliegue de una aplicación gráfica de escritorio, como LibreOffice y Firefox, pasando por un clásico despliegue de un servidor de web con WordPress, y cerrando el círculo con un ejemplo de Dockerfile para desplegar el código de Docker.

9.1 LibreOffice en un contenedor

Contenido del Dockerfile

```
FROM debian:stretch
MAINTAINER Jessie Frazelle <jess@linux.com>

RUN apt-get update && apt-get install -y \
    libreoffice \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

ENTRYPOINT [ "libreoffice" ]
```

Invocación en línea de órdenes

```
docker run -d \
  -v /etc/localtime:/etc/localtime:ro \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e DISPLAY=unix$DISPLAY \
  -v $HOME/slides:/root/slides \
  -e GDK_SCALE \
  -e GDK_DPI_SCALE \
  --name libreoffice \
  jess/libreoffice
```

Otra **alternativa**, sin necesidad de Dockerfile (de http://linoxide.com/how-tos/20-docker-containers-desktop-user/)

```
docker run \
-v $HOME/Documents:/home/libreoffice/Documents:rw \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e uid=$(id -u) -e gid=$(id -g) \
-e DISPLAY=unix$DISPLAY --name libreoffice \
chrisdaish/libreoffice
```

9.2 Firefox en un contenedor

Contenido del Dockerfile

```
FROM debian:sid
MAINTAINER Jessie Frazelle <jess@linux.com>

RUN apt-get update && apt-get install -y \
    dirmngr \
    gnupg \
    --no-install-recommends \
    && apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
0AB215679C571D1C8325275B9BDB3D89CE49EC21 \
    && echo "deb http://ppa.launchpad.net/mozillateam/firefox-next/ubuntu wily main" >>
/etc/apt/sources.list.d/firefox.list \
    && apt-get update && apt-get install -y \
    ca-certificates \
    firefox \
    hicolor-icon-theme \
    libasound2 \
    libgl1-mesa-dri \
    libgl1-mesa-glx \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

ENV LANG en-US

COPY local.conf /etc/fonts/local.conf

ENTRYPOINT [ "/usr/bin/firefox" ]
```

Archivo `local.conf`

```
<?xml version='1.0'?>
<!DOCTYPE fontconfig SYSTEM 'fonts.dtd'>
<fontconfig>
<match target="font">
<edit mode="assign" name="rgba">
<const>rgb</const>
</edit>
</match>
<match target="font">
<edit mode="assign" name="hinting">
<bool>true</bool>
</edit>
</match>
<match target="font">
<edit mode="assign" name="hintstyle">
<const>hintslight</const>
</edit>
</match>
<match target="font">
<edit mode="assign" name="antialias">
<bool>true</bool>
</edit>
</match>
<match target="font">
<edit mode="assign" name="lcdfilter">
<const>lcddefault</const>
</edit>
</match>
<match target="font">
<edit name="embeddedbitmap" mode="assign">
<bool>false</bool>
```

```
</edit>
</match>
</fontconfig>
```

9.3 (Otro) WordPress en un contenedor

Contenido del Dockerfile

```
FROM komljen/php-apache
MAINTAINER Alen Komljen <alen.komljen@live.com>

ENV WP_PASS aeshiethooghahtu4Riebooquae6Ithe
ENV WP_USER wordpress
ENV WP_DB wordpress
ENV APP_ROOT /var/www/html

ADD http://wordpress.org/latest.tar.gz wordpress.tar.gz

RUN \
  tar xzf wordpress.tar.gz -C ${APP_ROOT} --strip-components 1 && \
  rm wordpress.tar.gz

COPY start.sh start.sh

VOLUME ["$APP_ROOT"]

RUN rm /usr/sbin/policy-rc.d
CMD ["/start.sh"]

EXPOSE 80
```

Archivo `start.sh`

```
#!/usr/bin/env bash
#================================================================================
#
#    AUTHOR: Alen Komljen <alen.komljen@live.com>
#
#================================================================================
echo "Waiting for mysql:"
./tcp_wait.sh $MYSQL_PORT_3306_TCP_ADDR $MYSQL_PORT_3306_TCP_PORT
#--------------------------------------------------------------------------------
echo "Creating database:"
./create_db_mysql.sh $WP_DB $WP_USER $WP_PASS
#--------------------------------------------------------------------------------
sed -e "s/database_name_here/$WP_DB/
s/username_here/$WP_USER/
s/password_here/$WP_PASS/
s/localhost/$MYSQL_PORT_3306_TCP_ADDR/
/'AUTH_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'SECURE_AUTH_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'LOGGED_IN_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'NONCE_KEY'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'AUTH_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'SECURE_AUTH_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'LOGGED_IN_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/
/'NONCE_SALT'/s/put your unique phrase here/$(pwgen -c -n -1 65)/"             \
$APP_ROOT/wp-config-sample.php > $APP_ROOT/wp-config.php

chown -R www-data: $APP_ROOT
#--------------------------------------------------------------------------------
echo "Starting wordpress:"
source /etc/apache2/envvars
exec /usr/sbin/apache2 -D FOREGROUND
#================================================================================
```

9.4 Metaejemplo: Docker desde Docker

Podemos interpretarlo como "*la prueba definitiva*". En resumen se trata de instalar el entorno de desarrollo necesario para poder "compilar" Docker a partir del código fuente disponible en GitHub (con una cuenta *prestada*)

Contenido del ENORME Dockerfile:

```
FROM debian:jessie

# Add zfs ppa
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
E871F18B51E0147C77796AC81196BA81F6B0FC61 \
    || apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys
E871F18B51E0147C77796AC81196BA81F6B0FC61
RUN echo deb http://ppa.launchpad.net/zfs-native/stable/ubuntu trusty main >
/etc/apt/sources.list.d/zfs.list


# Allow replacing httpredir mirror
ARG APT_MIRROR=httpredir.debian.org
RUN sed -i s/httpredir.debian.org/$APT_MIRROR/g /etc/apt/sources.list

# Packaged dependencies
RUN apt-get update && apt-get install -y \
    apparmor \
    apt-utils \
    aufs-tools \
    automake \
    bash-completion \
    binutils-mingw-w64 \
    bsdmainutils \
    btrfs-tools \
    build-essential \
    clang \
    createrepo \
    curl \
    dpkg-sig \
    gcc-mingw-w64 \
    git \
    iptables \
    jq \
    libapparmor-dev \
    libcap-dev \
    libltdl-dev \
    libnl-3-dev \
    libprotobuf-c0-dev \
    libprotobuf-dev \
    libsqlite3-dev \
    libsystemd-journal-dev \
    libtool \
    mercurial \
    net-tools \
    pkg-config \
    protobuf-compiler \
    protobuf-c-compiler \
    python-dev \
    python-mock \
    python-pip \
    python-websocket \
    ubuntu-zfs \
    xfsprogs \
    libzfs-dev \
    tar \
```

```
    zip \
    --no-install-recommends \
    && pip install awscli==1.10.15
# Get lvm2 source for compiling statically
ENV LVM2_VERSION 2.02.103
RUN mkdir -p /usr/local/lvm2 \
    && curl -fsSL "https://mirrors.kernel.org/sourceware/lvm2/LVM2.${LVM2_VERSION}.tgz" \
            | tar -xzC /usr/local/lvm2 --strip-components=1
# See https://git.fedorahosted.org/cgit/lvm2.git/refs/tags for release tags

# Compile and install lvm2
RUN cd /usr/local/lvm2 \
    && ./configure \
            --build="$(gcc -print-multiarch)" \
            --enable-static_link \
    && make device-mapper \
    && make install_device-mapper
# See https://git.fedorahosted.org/cgit/lvm2.git/tree/INSTALL

# Configure the container for OSX cross compilation
ENV OSX_SDK MacOSX10.11.sdk
ENV OSX_CROSS_COMMIT a9317c18a3a457ca0a657f08cc4d0d43c6cf8953
RUN set -x \
    && export OSXCROSS_PATH="/osxcross" \
    && git clone https://github.com/tpoechtrager/osxcross.git $OSXCROSS_PATH \
    && ( cd $OSXCROSS_PATH && git checkout -q $OSX_CROSS_COMMIT) \
    && curl -sSL https://s3.dockerproject.org/darwin/v2/${OSX_SDK}.tar.xz -o
"${OSXCROSS_PATH}/tarballs/${OSX_SDK}.tar.xz" \
    && UNATTENDED=yes OSX_VERSION_MIN=10.6 ${OSXCROSS_PATH}/build.sh
ENV PATH /osxcross/target/bin:$PATH

# Install seccomp: the version shipped in trusty is too old
ENV SECCOMP_VERSION 2.3.1
RUN set -x \
    && export SECCOMP_PATH="$(mktemp -d)" \
    && curl -fsSL
"https://github.com/seccomp/libseccomp/releases/download/v${SECCOMP_VERSION}/libseccomp-
${SECCOMP_VERSION}.tar.gz" \
            | tar -xzC "$SECCOMP_PATH" --strip-components=1 \
    && ( \
            cd "$SECCOMP_PATH" \
            && ./configure --prefix=/usr/local \
            && make \
            && make install \
            && ldconfig \
    ) \
    && rm -rf "$SECCOMP_PATH"

# Install Go
# IMPORTANT: If the version of Go is updated, the Windows to Linux CI machines
#            will need updating, to avoid errors. Ping #docker-maintainers on IRC
#            with a heads-up.
ENV GO_VERSION 1.7.1
RUN curl -fsSL "https://storage.googleapis.com/golang/go${GO_VERSION}.linux-amd64.tar.gz" \
    | tar -xzC /usr/local

ENV PATH /go/bin:/usr/local/go/bin:$PATH
ENV GOPATH /go:/go/src/github.com/docker/docker/vendor

# Compile Go for cross compilation
ENV DOCKER_CROSSPLATFORMS \
    linux/386 linux/arm \
    darwin/amd64 \
    freebsd/amd64 freebsd/386 freebsd/arm \
    windows/amd64 windows/386
```

```
# Dependency for golint
ENV GO_TOOLS_COMMIT 823804e1ae08dbb14eb807afc7db9993bc9e3cc3
RUN git clone https://github.com/golang/tools.git /go/src/golang.org/x/tools \
    && (cd /go/src/golang.org/x/tools && git checkout -q $GO_TOOLS_COMMIT)

# Grab Go's lint tool
ENV GO_LINT_COMMIT 32a87160691b3c96046c0c678fe57c5bef761456
RUN git clone https://github.com/golang/lint.git /go/src/github.com/golang/lint \
    && (cd /go/src/github.com/golang/lint && git checkout -q $GO_LINT_COMMIT) \
    && go install -v github.com/golang/lint/golint

# Install CRIU for checkpoint/restore support
ENV CRIU_VERSION 2.2
RUN mkdir -p /usr/src/criu \
    && curl -sSL https://github.com/xemul/criu/archive/v${CRIU_VERSION}.tar.gz | tar -v -C
/usr/src/criu/ -xz --strip-components=1 \
    && cd /usr/src/criu \
    && make \
    && make install-criu

# Install two versions of the registry. The first is an older version that
# only supports schema1 manifests. The second is a newer version that supports
# both. This allows integration-cli tests to cover push/pull with both schema1
# and schema2 manifests.
ENV REGISTRY_COMMIT_SCHEMA1 ec87e9b6971d831f0eff752ddb54fb64693e51cd
ENV REGISTRY_COMMIT 47a064d4195a9b56133891bbb13620c3ac83a827
RUN set -x \
    && export GOPATH="$(mktemp -d)" \
    && git clone https://github.com/docker/distribution.git
"$GOPATH/src/github.com/docker/distribution" \
    && (cd "$GOPATH/src/github.com/docker/distribution" && git checkout -q "$REGISTRY_COMMIT") \
    && GOPATH="$GOPATH/src/github.com/docker/distribution/Godeps/_workspace:$GOPATH" \
            go build -o /usr/local/bin/registry-v2 github.com/docker/distribution/cmd/registry \
    && (cd "$GOPATH/src/github.com/docker/distribution" && git checkout -q "$REGISTRY_COMMIT_SCHEMA1")
\
    && GOPATH="$GOPATH/src/github.com/docker/distribution/Godeps/_workspace:$GOPATH" \
            go build -o /usr/local/bin/registry-v2-schema1 github.com/docker/distribution/cmd/registry
\
    && rm -rf "$GOPATH"

# Install notary and notary-server
ENV NOTARY_VERSION v0.3.0
RUN set -x \
    && export GOPATH="$(mktemp -d)" \
    && git clone https://github.com/docker/notary.git "$GOPATH/src/github.com/docker/notary" \
    && (cd "$GOPATH/src/github.com/docker/notary" && git checkout -q "$NOTARY_VERSION") \
    && GOPATH="$GOPATH/src/github.com/docker/notary/vendor:$GOPATH" \
            go build -o /usr/local/bin/notary-server github.com/docker/notary/cmd/notary-server \
    && GOPATH="$GOPATH/src/github.com/docker/notary/vendor:$GOPATH" \
            go build -o /usr/local/bin/notary github.com/docker/notary/cmd/notary \
    && rm -rf "$GOPATH"

# Get the "docker-py" source so we can run their integration tests
ENV DOCKER_PY_COMMIT e2655f658408f9ad1f62abdef3eb6ed43c0cf324
RUN git clone https://github.com/docker/docker-py.git /docker-py \
    && cd /docker-py \
    && git checkout -q $DOCKER_PY_COMMIT \
    && pip install -r test-requirements.txt

# Set user.email so crosbymichael's in-container merge commits go smoothly
RUN git config --global user.email 'docker-dummy@example.com'

# Add an unprivileged user to be used for tests which need it
RUN groupadd -r docker
RUN useradd --create-home --gid docker unprivilegeduser
```

```
VOLUME /var/lib/docker
WORKDIR /go/src/github.com/docker/docker
ENV DOCKER_BUILDTAGS apparmor pkcs11 seccomp selinux

# Let us use a .bashrc file
RUN ln -sfv $PWD/.bashrc ~/.bashrc
# Add integration helps to bashrc
RUN echo "source $PWD/hack/make/.integration-test-helpers" >> /etc/bash.bashrc

# Register Docker's bash completion.
RUN ln -sv $PWD/contrib/completion/bash/docker /etc/bash_completion.d/docker

# Get useful and necessary Hub images so we can "docker load" locally instead of pulling
COPY contrib/download-frozen-image-v2.sh /go/src/github.com/docker/docker/contrib/
RUN ./contrib/download-frozen-image-v2.sh /docker-frozen-images \
    buildpack-deps:jessie@sha256:25785f89240fbcdd8a74bdaf30dd5599a9523882c6dfc567f2e9ef7cf6f79db6 \
    busybox:latest@sha256:e4f93f6ed15a0cdd342f5aae387886fba0ab98af0a102da6276eaf24d6e6ade0 \
    debian:jessie@sha256:f968f10b4b523737e253a97eac59b0d1420b5c19b69928d35801a6373ffe330e \
    hello-world:latest@sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
# See also "hack/make/.ensure-frozen-images" (which needs to be updated any time this list is)

# Install tomlv, runc, containerd and grimes
# Please edit hack/dockerfile/install-binaries.sh to update them.
COPY hack/dockerfile/install-binaries.sh /tmp/install-binaries.sh
RUN /tmp/install-binaries.sh tomlv runc containerd grimes

# Wrap all commands in the "docker-in-docker" script to allow nested containers
ENTRYPOINT ["hack/dind"]

# Upload docker source
COPY . /go/src/github.com/docker/docker
```

Invocación desde la línea de órdenes

1. Ensamblar todo el entorno de desarrollo. La primera vez es muy lento.

```
docker build -t docker .
```

2. Para probar con comodidad, montamos nuestro fuente en un contenedor interactivo:

```
docker run -v `pwd`:/go/src/github.com/docker/docker --privileged -i -t docker bash
```

3. Ejecutamos las pruebas

```
docker run --privileged docker hack/make.sh test-unit test-integration-cli test-
docker-py
```

4. Publicamos el resultado de nuevo en GitHub:

```
docker run --privileged \
 -e AWS_S3_BUCKET=baz \
 -e AWS_ACCESS_KEY=foo \
 -e AWS_SECRET_KEY=bar \
 -e GPG_PASSPHRASE=gloubiboulga \
 docker hack/release.sh
```

10 APÉNDICE. DOCKER-COMPOSE.YML

Este anexo **no** es exhaustivo, y `docker-compose` se encuentra activamente **en evolución**, de manera que la documentación original es una fuente irrenunciable de información y referencia.

El archivo `docker-compose.yml` especifica las características para el despliegue mediante `docker-compose` de una aplicación distribuida. Su contenido son líneas de texto que se ajustan a una sintaxis especificada a partir de YAML (`https://docs.docker.com/compose/yml/`)

Cada uno de los servicios que se definan en `docker-compose.yml` debe especificar exactamente una imagen. El resto de claves son opcionales, y son análogas a sus correspondientes para la orden `run` de Docker. Las órdenes incluidas en el Dockerfile no necesitan repetirse en `docker-compose.yml`.

10.1 Breve introducción a YAML

YAML (*YAML Ain't Markup Language*) es un lenguaje de marcas que sirve para describir estructuras con propiedades (árboles de objetos) mediante un fichero de texto de extensión yml. Estos archivos son más fáciles de leer que JSON (más información en **Wikipedia** `https://en.wikipedia.org/wiki/YAML`)

Para construir ficheros de este tipo puede emplearse cualquier editor de texto, si bien algunos como Visual Studio Code reconocen este lenguaje y prestan alguna ayuda.
Para disponer de un parser, se puede optar por:

- `npm install js-yaml`, que instalará un parser local
- `https://github.com/nodeca/js-yaml`, que es un servicio online

En cualquier caso, para nuestros objetivos, lo interesante es validarlo de manera que corresponda no solo con YAML en general, sino con el caso concreto de Docker Compose (validación online en `https://lorry.io/`)

Las reglas en un fichero YAML no son complicadas, pero merece la pena destacar:

- No pueden usarse tabuladores, tan solo espacios en blanco.
- Las propiedades y listas se deben indentar con un espacio o más.
- Las mayúsculas y minúsculas son significativas tanto en los identificadores de propiedades como en los de claves.

Un ejemplo mínimo

```
- login: ceronin23
  pass: kjh_47!
  dni: 5566557
- login: jascua
  pass: Cko05Hs
```

Se trata simplemente de una relación de claves, que podrían anidarse si fuera necesario. Muchos parsers podrían tratar el valor `5566557` como numérico. No es obligatorio el uso de comillas en las cadenas.

Los símbolos & y * se usan para definir alias y referenciarlos, de manera que algunos ficheros de configuración se estructuran mejor y se simplifican:

```
defaults: &defaults
  adapter:  postgres
  host:     localhost

development:
  database: myapp_development
  <<: *defaults

test:
  database: myapp_test
  <<: *defaults
```

& identifica el enlace ("`defaults`"), << se interpreta como "*lugar donde incluir la clave*", y * realiza la inclusión. La versión expandida del ejemplo anterior sería:

```
defaults:
  adapter:  postgres
  host:     localhost

development:
  database: myapp_development
  adapter:  postgres
  host:     localhost

test:
  database: myapp_test
  adapter:  postgres
  host:     localhost
```

Flow style

Hay una sintaxis alternativa en YAML denominada "*flow style*", en la cual se evita la indentación usando corchetes y llaves, para vectores y claves respectivamente. Por esta característica, JSON puede interpretarse como un subconjunto de YAML.

```
---
# Vectores
colors:
  - red
  - blue
# con flow style...
colors: [red, blue]

# Claves
- name: Xavier
  age: 24
# con flow style...
- {name: Xavier, age: 24}
```

10.2 image

Identificador de la imagen, local o remota. Composer intentará obtenerla (*pull*) si no se dispone de ella localmente.

```
image: ubuntu
```

```
image: orchardup/postgresql
image: a4bc65fd
```

10.3 build

Ruta a un directorio con el Dockerfile. Si se trata de una ruta relativa, lo será respecto a la del archivo yml. Compose lo construirá y designará con un nombre.

```
build: ./dir

build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

10.3.1 dockerfile

Dockerfile alternativo para construir la imagen.

10.4 command

Sustituye a la orden por defecto.

```
command: bundle exec thin -p 3000
```

10.5 links

Enlaza con contenedores de otro servicio. Especifica el nombre del servicio y el alias para el `link` (SERVICIO:ALIAS), o solo el nombre si se va a mantener el mismo en el alias.

```
web:
  links:
 -   db
 -   db:database
 -   redis
```

El nombre del servicio actúa como nombre DNS del servidor salvo que se haya definido un alias. Los enlaces expresan dependencias entre servicios, e influyen en el orden de arranque del servicio.

10.6 external_links

Enlaces a contenedores externos a este compose.yml, o incluso externos a Compose, especialmente para los que ofrecen servicios compartidos. La semántica de esta orden es similar a `links` cuando se usa para especificar el nombre del contenedor y el alias del enlace (CONTENEDOR:ALIAS).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

10.7 ports

Puertos expuestos, bien indicando ambos (EQUIPO:CONTENEDOR), o bien indicando únicamente el del equipo contenedor (se tomará un puerto del anfitrión al azar).

Nota: Se recomienda especificar los puertos y su correspondencia como cadenas con comillas.

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

10.8 expose

Expone puertos sin publicarlos al equipo anfitrión, de manera que únicamente tendrán acceso los servicios enlazados dentro de Docker. Solo se puede indicar el puerto interno.

```
expose:
 - "3000"
 - "8000"
```

10.9 volumes

Monta rutas como volúmenes, especificando opcionalmente una ruta en el anfitrión (HOST:CONTENEDOR), o un modo de acceso (HOST:CONTENEDOR:ro).

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

Se puede montar una ruta relativa al anfitrión, que será expandida de forma relativa al directorio del archivo de configuración de Compose que se esté empleando. Las rutas relativas comienzan siempre por `.` o `..`

10.10 environment

Añade variables de entorno

Los valores de las variables de entorno que solo disponen de una clave se calculan en la máquina en la que se ejecuta Compose, siendo especialmente útil para valores secretos o a medida del anfitrión.

```
environment:
  RACK_ENV: development
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

10.11 labels

Añade metadatos a contenedores mediante `labels` de Docker. Puede elegirse entre array y diccionario.

```
labels:
```

```
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

10.12 container_name

Establece un nombre para el contenedor.

```
container_name: my-web-container
```

Como los nombres de contenedores han de ser únicos, no puede (daría error) escalarse un servicio si se ha explicitado un nombre específico para su contenedor.

10.13 dns

Servidores DNS a utilizar.

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

10.14 Análogas a docker run

working_dir, entrypoint, user, hostname, domainname, mac_address, mem_limit, memswap_limit, privileged, restart, stdin_open, tty, cpu_shares, cpuset, read_only, volume_driver

Cada orden tiene un único parámetro, como el correspondiente en `docker run`.

```
cpu_shares: 73
cpuset: 0,1

working_dir: /code
entrypoint: /code/entrypoint.sh
user: postgresql

hostname: foo
domainname: foo.com

mac_address: 02:42:ac:11:65:43

mem_limit: 1000000000
memswap_limit: 2000000000
privileged: true

restart: always

stdin_open: true
tty: true
read_only: true

volume_driver: mydriver
```

11 APÉNDICE. EJERCICIOS

11.1 Pequeño test

1. Se pretende desplegar un componente web utilizando un contenedor Docker. El código fuente del componente se halla en el directorio **/usr/aplicacion/componentes/web.** También se encuentra en dicho directorio el Dockerfile con el que se creará la imagen del componente. En el Dockerfile del componente web se utiliza el comando **EXPOSE 8000** para indicar que el servidor será accesible a través del puerto 8000 del contenedor.

Indica que secuencia de comandos utilizarías para realizar dicho despliegue si suponemos que estamos situados en el directorio **/usr/aplicacion/componentes,** y queremos acceder al componente a través del puerto 80 del host anfitrión.

a)

```
$ cd  web
$ docker build
$ docker run  -p  80:8000 web_image
```

b)

```
$ cd  web
$ docker build -t web_image .
$ docker run  -p  8000:80 image
```

c)

```
$ cd  web
$ docker build -t web_image .
$ docker run  -port 8000 web_image
```

d)

```
$ cd web
$ docker build -t  web_image  .
$ docker run  -p  80:8000 web_image
```

2. **El componente web de la pregunta anterior está en ejecución!!** dentro de un contenedor con identificador **"ef5jf678rlg…."**. Éste se ha construido a partir de la imagen **web_image.** ***Hemos realizado una modificación en el código de dicho componente web y pretendemos crear una nueva imagen **web_image.**

Suponemos que estamos situados en el directorio **usr/aplicacion/componentes/web** (recordamos que el Dockerfile del componente se halla en dicho directorio).
Indica una secuencia válida de comandos para lanzar de nuevo el componente modificado asegurándote que el anterior componente esté eliminado:

a)

```
$ docker rm –f ef5j
$ docker build –t web_image .
```

b)

```
$ docker rmi -f web_image
$ docker build –t web_image .
```

```
$ docker run ……
```

c)

```
$ docker rm –f ef5j
$ docker rmi -f web_image
$ docker build –t web_image .
$ docker run ……
```

d)

```
$ docker rm -f web_image
$ docker build –t web_image .
$ docker run ……
```

3. La imagen del componente web con el que estamos trabajando contiene node, la librería binaria de zmq y también npm. Sin embargo no tiene instalado el *binding* para node de zmq. Pretendemos construir una nueva imagen del componente que permita utilizar la librería zmq desde node.

Indica cuál de las siguientes afirmaciones es correcta:

a) La única forma de hacerlo es modificando el Dockerfile del componente y volviendo a crear la imagen del componente con "**docker build** …"

b) Se podría ejecutar el contenedor en modo interactivo con "**docker run -i -t ……"** e instalar zmq con "**npm install zmq"** directamente en la máquina virtual docker. Posteriormente se podría obtener la nueva imagen con "**docker commit …"** a partir del contenedor en ejecución.

c) Se podría ejecutar el contenedor en modo interactivo con "**docker run -i -t ….."** e instalar zmq con "**npm install zmq"** directamente en la máquina virtual docker. La imagen quedaría actualizada automáticamente.

d) Si utilizamos un Dockerfile para obtener la nueva imagen, éste debería contener el comando **"CMD npm install zmq".**

4. Ahora le añadimos al componente web un componente de base de datos. Pretendemos utilizar docker-compose para desplegar el nuevo servicio. El fichero **docker-compose.yml** se halla en el directorio **/usr/aplicacion/servicio.** Suponemos que estamos situados en este directorio del host anfitrión.

Pretendemos:

1. Lanzar el servicio con todos los componentes.
2. Comprobar que el servicio está en marcha.
3. Parar el servicio con todos los componentes.
4. Eliminar todos los contenedores asociados al servicio.

Indica cuál de las siguientes afirmaciones es cierta:

a) La siguiente secuencia permite realizar las cuatro acciones especificadas:

```
$ docker-compose up
$ docker ps -a
$ docker-compose stop
$ docker-compose rm -f
```

b) La siguiente secuencia permite realizar las cuatro acciones especificadas:

```
$ docker-compose up
$ docker ps -a
$ docker stop
$ docker rm -f
```

c) La forma habitual de eliminar los contenedores asociados al servicio es borrando, individualmente, cada uno de ellos.

d) Para realizar el despligue con **"docker-compose up"** es necesario que estén creadas todas las imágenes de los componentes previamente.

11.1.1 Soluciones

1. d)

2. c)

3. b)

4. a)

11.2 TcpProxy para WordPress

En el ejemplo de WordPress sobre Fedora del apartado 6.5 se describe, configura y despliega un servicio **WordPress**.

Una vez desplegado este servicio, se solicita la adaptación e inclusión de un componente adicional procedente del myTCPProxy.js de la práctica 1. Aquí se ofrece como solución el siguiente **Lab3myTcpProxyA.js**

Código (Lab3myTcpProxyA.js)

```
var net = require('net');

var LOCAL_PORT = 80;
var LOCAL_IP = '127.0.0.1';
if (process.argv.length != 4) {
  console.error("The WordPress port and IP should be given as arguments!!");
  process.exit(1);
}
var REMOTE_PORT = process.argv[2];
var REMOTE_IP = process.argv[3];

var server = net.createServer(function (socket) {
  socket.on('data', function (msg) {
    var serviceSocket = new net.Socket();
    serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
      serviceSocket.write(msg);
    });
    serviceSocket.on('data', function (data) {
      socket.write(data);
    });
```

```
  });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Pero disponer de este código no es suficiente para integrar el nuevo componente en un despliegue del servicio WordPress. El componente adicional debe intermediar en las peticiones de servicio, de manera que actúe como receptor de las mismas y las reenvíe al puerto e IP del componente web de WordPress. Para ello **se pide completar los siguientes aspectos**:

1. Crear el **Dockerfile** necesario para desplegar este componente. Debe reenviar las peticiones a la IP y puerto de WordPress. Como referencia puedes *fijarte* en el siguiente Dockerfile usado para el componente **worker** en el sistema client-broker-worker:

```
FROM tsr/fedora-node-devel
COPY ./worker.js worker.js
# We assume that each worker is linked to the broker
# container. Such broker container should have the
# 'BROKER' name and should be linked to this worker.
CMD node worker $BROKER_PORT
```

2. Nuevo **docker-compose.yml** en el que se incorpore este componente

11.2.1 Solución

Es importante observar que la aplicación distribuida ha de mantener las mismas interfaces externas, de manera que el puerto 8000, anteriormente asociado al componente web, ahora deberá vincularse al proxy TCP. Esto NO obliga a cambiar el puerto del componente web, sino su visibilidad (p.ej. mediante `expose`) como acceso al servicio en el fichero de despliegue docker-compose.yml.

Dockerfile

Posee una parte general análoga al ejemplo anterior (worker) sobre la que tendremos que aplicar algunos ajustes:

a) El programa ***js*** a ejecutar, que hay que copiar previamente.
b) Hacer accesible el puerto de servicio al que atenderá este proxy TCP
c) Colocar los parámetros adecuados del componente **web** (puerto e IP) para que sean transmitidos al programa en su invocación.

```
FROM tsr/fedora-node-devel
COPY ./Lab3myTcpProxy.js Lab3myTcpProxy.js
EXPOSE 80
CMD node Lab3myTcpProxy $WEB_PORT $WEB_ADDR
```

Es importante usar la orden adecuada para crear el componente a partir de este Dockerfile, dado que en ella se establece el nombre del componente. Para ello, en el directorio en que se encuentra tanto el Dockerfile como el archivo Lab3myTcpProxy.js, ejecutamos:

```
docker build -t proxy .
```

docker-compose.yml

Necesitamos una sección para el componente proxy, destacando:

a) Asociación con el componente que hemos creado (orden `image`)
b) Acceso a la información creada para el componente `web` en su despliegue (valor `web` para la orden `links`)
c) Registro para usar un puerto (8000) del anfitrión como si se tratara del 80 (variable `LOCAL_PORT`) del contenedor (valor "8000:80" para la orden `ports`)

Y también necesitamos "ocultar" el puerto 8000 del componente `web` para impedir que colisione con el del proxy, y/o que pueda ser accedido directamente sin pasar por el proxy.

a) Retirar orden `ports` de la sección `web`
b) Colocar valor "8000" para la orden `expose` en la sección `web` (permite que sea utilizado internamente)

```
proxy:
  image: proxy
  links:
    - web
  ports:
    - "8000:80"
  environment:
    - WEB_PORT=8000
    - WEB_ADDR=web
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  expose:
    - "8000"
  links:
    - db
  volumes:
    - .:/code
db:
  image: orchardup/mysql
  environment:
    MYSQL_DATABASE: wordpress
```

Si se han seguido los pasos anteriores, y nos colocamos en el directorio que contiene este fichero de configuración, para hacer funcionar la magia y arrancar una instancia de cada uno de los 3 componentes ejecutaremos:

```
docker-compose up
```

11.3 Publicador/Subscriptor

Este ejercicio gira en torno a una aplicación distribuida basada en NodeJS+ZMQ y su ejecución en contenedores de una máquina virtual del portal.

Se pretende emplear el patrón PUB/SUB para comunicar los componentes. Únicamente se cuenta con un proceso publicador (`pubextra.js`) que enviará a intervalos regulares un mensaje de un tema, mientras que el número de suscriptores (`subextra.js`) es indeterminado aunque todos (los suscriptores) comparten el mismo código.

Un ejemplo de invocación de `pubextra.js` es:

```
node pubextra.js tcp://*:9999 2 deporte ciencia sociedad
```

que, en este ejemplo, debería interpretarse como:

- El punto de entrada (admitir peticiones) del servicio es `tcp://*:9999` (primer argumento)
- El tiempo transcurrido entre publicaciones de mensajes es **2** segundos (segundo argumento). Tras cada envío deberá escribir un aviso con el nombre del tópico en pantalla.
- Los temas (resto de argumentos) entre los que hay que *conmutar* son `deporte` (primer mensaje), `ciencia` (segundo) y `sociedad` (tercero). Tras el último se debe volver al primero. No hay un número de temas preestablecido.

Una implementación de **pubextra.js** que encaja con esta descripción:

```
// pubextra.js
var zmq = require('zmq')
var publisher = zmq.socket('pub')

// Check how many arguments have been received.
if (process.argv.length < 5) {
  console.error("Format is 'node pubextra URL secs topics+'");
  console.error("Example: 'node pubextra tcp://*:9999 2 deporte ciencia sociedad'");
  process.exit(1);
}
// Get the connection URL.
var url = process.argv[2];
// Get period
var period = process.argv[3];
// Get topics
var topics = process.argv.slice(4, process.argv.length);
var i=0;
var count=0;
publisher.bind(url, function(err) {
  if(err) console.log(err);
  else console.log('Listening on '+url+' ...');
})

setInterval(function() {
  ++count;
  publisher.send(topics[i]+' msg '+count);
  console.log('Sent '+topics[i]+' msg '+count);
  i=(i+1)%topics.length;
  if (count>100) process.exit();
},period*1000);
```

En el caso del/los suscriptor/es, la invocación de `subextra.js` es:

```
node subextra.js tcp://localhost:9999 deporte
```

que, en este ejemplo, debería interpretarse como:

- Conectar con el publicador en `tcp://localhost:9999` (primer argumento)
- Desea recibir mensajes únicamente del tópico `deporte` (segundo argumento). Tras cada recepción, el suscriptor debe emitir un aviso reproduciendo el mensaje recibido.

Una implementación de **subextra.js** que encaja con esta descripción:

```
// subextra.js
var zmq = require('zmq')
```

```
var subscriber = zmq.socket('sub')

// Check how many arguments have been received.
if (process.argv.length != 4) {
  console.error("Format is 'node subextra URL topic'");
  console.error("Example: 'node subextra localhost:9999 deporte'");
  process.exit(1);
}
// Get the connection URL.
var url = process.argv[2];
// Get topic
var topic = process.argv[3]
subscriber.on('message', function(data) {
  console.log('Received '+data);
})

subscriber.connect(url);
subscriber.subscribe(topic);
```

Un posible uso de estos componentes puede constar de una instancia del publicador (iníciala al final) y tres instancias de los suscriptores. Se recomienda ejecutar cada uno en una ventana diferente del *shell*:

```
node subextra.js tcp://localhost:9999 moda
node subextra.js tcp://localhost:9999 salud
node subextra.js tcp://localhost:9999 ocio
node pubextra.js tcp://*:9999 1 moda salud negocios ocio
```

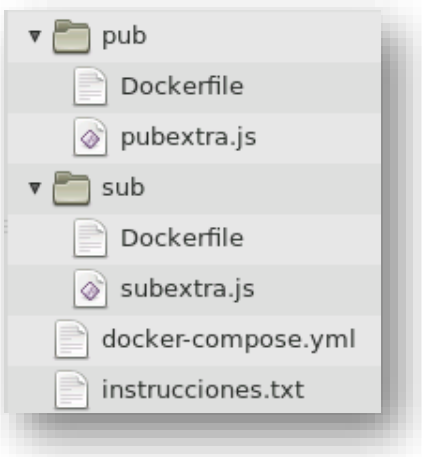
Dada ya la descripción y código de los componentes, se desea desplegarlos en contenedores de nuestra máquina virtual, constituyendo una aplicación distribuida.
Para ello deberás elaborar un fichero de configuración `Dockerfile` para el publicador, y otro compartido por los suscriptores.
Los requisitos básicos *incluyen* estas instrucciones:

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel npm make
RUN npm install zmq
```

Pero cada componente deberá añadir sus requisitos específicos para ser desplegado.

- Los suscriptores siempre solicitarán el tópico `ocio` (dato desconocido para el publicador).
- El publicador siempre atenderá a `tcp://*:8888` (dato desconocido para los suscriptores).

Debe desarrollarse el `Dockerfile` de cada uno, el `docker-compose.yml` que relacione estos componentes, y ejecutar el escalado a 4 suscriptores.



Para realizar este ejercicio es conveniente crear un directorio pub que contenga todo el material para el despliegue del publicador, un directorio sub para el relacionado con el suscriptor, mientras que el archivo `docker-compose.yml` debe encontrarse directamente en el directorio raíz.

Es buena idea crear un archivo `instrucciones.txt` que indique con exactitud la secuencia de órdenes necesarias para construir los componentes y para desplegar, tal y como ya se ha dicho, una aplicación distribuida compuesta por 1 publicador y 4 suscriptores según los detalles indicados.

11.3.1 Solución

instrucciones.txt

Suponiendo que `pubextra.js` se encuentra en pub, que `subextra.js` se encuentra en sub, y que nuestro directorio actual es el raíz.

```
#!/bin/sh
for i in pub sub
do
  cd $i
  docker build  -t ${i}extra .
  cd ..
done
docker-compose up
docker-compose scale pub=1 sub=4
```

docker-compose.yml

```
version: '2'
services:
  sub:
    image: subextra
    links:
      - pub
    environment:
     - PUB_URL=tcp://pub:8888

  pub:
    image: pubextra
```

pub/Dockerfile

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel npm make
RUN npm install zmq
COPY ./pubextra.js pubextra.js
EXPOSE 8888
CMD node pubextra.js tcp://*:8888 1 moda salud deportes ocio
```

sub/Dockerfile

```
FROM fedora
RUN dnf -y -q install nodejs zeromq-devel npm make
RUN npm install zmq
COPY ./subextra.js subextra.js
# We assume that each subscriber is linked to the publisher
# container. Such publisher container should have the
# 'PUBEXTRA' name and should be linked to this subscriber.
CMD node subextra.js $PUB_URL ocio
```