TSR - PRÁCTICA 4

DESPLIEGUE

Esta práctica 4 se desarrollará a lo largo de tres sesiones. Su objetivo principal es:

> "Comprender los retos que conlleva el despliegue de un servicio multi-componente."

Esta práctica requiere conocimientos asociados al tema 4 (**Despliegue**), y depende parcialmente de la práctica 3, **ØMQ**, especialmente en lo relativo al sistema *client-broker-worker* (**cbw**) con socket ROUTER-ROUTER, variante tolerante a fallos.

Debido al uso de Docker, la práctica solo puede ser completada en instalaciones tales como las virtuales de portal (DSIC) o la imagen de VirtualBox disponible desde el inicio de este curso. En todo caso, es necesario un sistema con Docker instalado, node y zeromq, versión 5.

Las actividades se basan en un material (`tsr_lab4_material.zip`) que, al descomprimirse, dará lugar a varias carpetas. Dentro de este documento se mencionan archivos contenidos en dichas carpetas. En cada sesión encontraremos apartados con instrucciones seguidos de otros con cuestiones sobre los resultados.

La **primera sesión** se refiere al manejo básico de Docker, y afecta a la generación de las diferentes imágenes necesarias para el despliegue del sistema **cbw** anteriormente mencionado.

En la **segunda sesión** se desea añadir un par de componentes al sistema anterior. En primer lugar un componente *logger* que nos obligará a acceder al almacenamiento persistente del anfitrión, y en segundo lugar añadiremos un *cliente externo* que se ejecutará desde un equipo exterior que tenga node y zeromq, pero debe conectar con el bróker desplegado.

La **tercera sesión** propone el despliegue de sistemas preconfigurados, discutiendo los casos en los que esta posibilidad pueda ser recomendable, y completando una instalación junto con la verificación de su funcionamiento.

Para esta práctica se necesita:

1. Los conocimientos necesarios acerca de NodeJS y ZeroMQ que han sido objeto de estudio hasta la fecha. Esta práctica se encuentra específicamente ligada al tema 4, **Despliegue de servicios**, donde se hace hincapié en la tecnología de contenerización que representa Docker.
2. Los materiales accesibles en PoliformaT (`tsr_lab4_material.zip`) en el directorio correspondiente a la tercera práctica.
3. Una **máquina virtual de portal**, que contiene ya una instalación de Docker y permite realizar los ejercicios planteados en esta práctica. En caso de que el servicio (demonio) `docker` no se encuentre en marcha, deberás ejecutar:

```
sudo systemctl start docker
```

1 Sesión 1. Primeros pasos con Docker. Despliegue de CBW

Para empezar la sesión de prácticas, vamos a crear “a mano” y ejecutar “a mano” los contenedores de nuestra aplicación distribuida. En una situación realista este proceso no se realiza “a mano”, sino que se cuenta con herramientas para automatizar el proceso. Sin embargo, es importante conocer qué debe realizarse en cada etapa, de forma que comprendamos mejor para qué nos ayudan las herramientas de automatización.

1.1 Construyendo la imagen base con Ubuntu, NodeJS y ØMQ

Necesitamos el punto de partida con el que estamos familiarizados, pero aplicado a nuestros contenedores. En nuestro caso queremos un sistema Linux que hayamos probado, con node y zeromq en su versión 5. En el material de teoría del tema 4 se detalla cómo generar esta imagen inicial a la que llamaremos `tsr-zmq`. Te incluimos el Dockerfile y la orden indicados.

Dispones de este mismo Dockerfile en la carpeta “tsr-zmq”, dentro de la carpeta CBW.

1.1.1 Dockerfile

```
FROM ubuntu:22.04
WORKDIR /root
RUN apt-get update -y
RUN apt-get install curl ufw gcc g++ make gnupg -y
RUN curl -sL https://deb.nodesource.com/setup_20.x | bash -
RUN apt-get update -y
RUN apt-get install nodejs -y
RUN apt-get upgrade -y
RUN npm init -y
RUN npm install zeromq@5
```

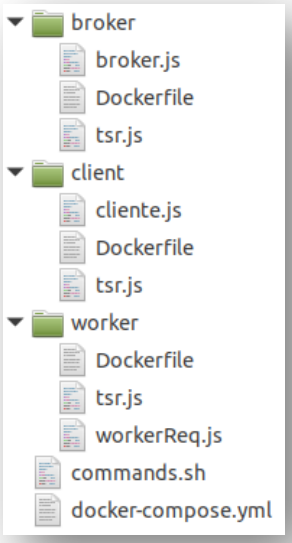
1.1.2 Orden

```
docker build -t tsr-zmq .
```

Con esta orden habremos creado la imagen “tsr-zmq”.

Comprueba que la imagen existe tras ejecutar la orden, situándote en un terminal en la carpeta adecuada.

1.2 Despliegue manual de las imágenes individuales de CBW



Ahora debemos generar las imágenes de los tres componentes: una imagen para el bróker, otra para el trabajador y otra para el cliente.

El cliente y los trabajadores se han modificado ligeramente para que incluyan automáticamente un identificador derivado del nombre del host. Este detalle ya está implementado, y resulta conveniente dado que queremos emplear la misma orden para todos los clientes y trabajadores, sin que necesitemos indicar para cada uno de ellos un "ID" diferente. Este detalle se logrará indicando como ID el valor especial "HN" como argumento tanto a clientes como a trabajadores

En la ilustración de la derecha vemos el contenido de la carpeta "CBW": cada componente cuenta con su propia carpeta, con Dockerfile, código fuente y los archivos que necesita.

Paso 1: generar una imagen para el bróker.

Abre un terminal en la carpeta "CBW/broker" y en ese terminal crea la imagen, que debes crear con el nombre "imbroker".

Cuando hayas generado la image, ejecútala con la orden:

```
docker run imbroker
```

Paso 2: averiguar la dirección IP del broker.

Una vez has completado con éxito el paso 1, tienes en ejecución un bróker contenerizado. Este contenedor es un ordenador completo que se encuentra en ejecución. Para que los clientes y trabajadores conecten con él, necesitan su dirección IP.

Abre otro terminal y en él debes ejecutar la orden "`docker inspect`" para averiguar la IP del broker. Revisa el material de teoría para asegurarte de emplear la orden correctamente.

La orden "`docker inspect`" te ofrece mucha información del contenedor. Entre toda esa información debes encontrar la dirección IP.

```
"172.17.0.2"
```

Paso 3: Adaptar los “Dockerfile”.

Los ficheros Dockerfile de cliente y worker necesitan la dirección IP del bróker para poder contactar con él.

Edita ambos Dockerfile, y actualízalos para que la orden “node” tenga el parámetro correcto como dirección IP del bróker.

Paso 4: Crear las imágenes para los contenedores de cliente y worker.

Abre un terminal, empleando la carpeta adecuada, y ahí crea una imagen para el cliente al que llamaremos “imclient” otra imagen para el worker, al que llamaremos “imworker”.

Paso 5: Ejecutar al menos un cliente y un worker

Cada uno de ellos en un terminal diferente, utiliza la orden “docker run” para ejecutar al menos un cliente y un trabajador. Puedes poner más contenedores en marcha con más terminales.

Paso 5: Comprobación

Asegúrate de que tienes creadas correctamente las imágenes, que los contenedores están en marcha y observa que las peticiones efectuadas por los clientes han sido atendidas por el trabajador correspondiente.

1.3 Despliegue automatizado del sistema CBW

*En el material de teoría del tema 4, transparencias y guía del alumno del tema 4 se menciona cómo construir un despliegue orquestado de varios componentes para crear una aplicación (**cbw**) distribuida. Revisa este material.*

El despliegue manual de una aplicación distribuida no es adecuado conforme nos acercamos a un ámbito más realista.

Para ayudarnos debemos emplear herramientas de automatización. En esta parte de la práctica emplearemos la herramienta "docker compose".

Edita el fichero "docker-compose.yml", disponible como parte del material para esta práctica en la carpeta CWB Observa cómo este fichero facilita la inyección de dependencias, en nuestro caso, la dirección IP del broker, mediante simples variables de entorno.

Paso 1: Edita los ficheros "Dockerfile" y sustituye la dirección IP que actualizaste a mano en la sección anterior, por la variable de entorno $BROKER_HOST. Esta variable es la que se emplea en "docker-compose.yml"

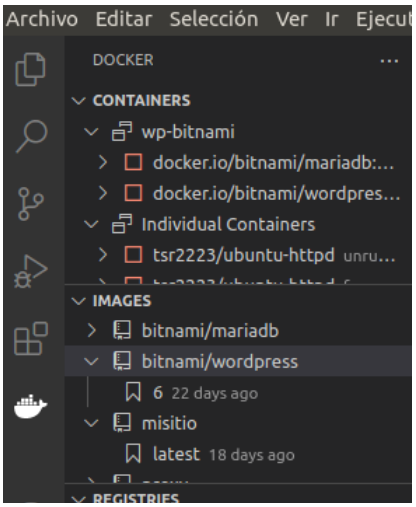
Paso 2: Elimina las versiones anteriores de las imágenes "imworker" e "imclient". La herramienta de automatización no actualiza las imágenes si ya existen. Pero las crea en caso de no existir. En todo caso eliminar las imágenes nos permite ejercitar las órdenes docker "rmi" (remove image) y "rm" (remove container). El argumento especial "-f" permite eliminar imágenes aun teniendo contenedores en funcionamiento. También deberás detener las componentes desplegadas. Emplea para ello la orden "docker stop"

Despliega 2 clientes y 4 trabajadores. Para ello emplea un terminal que habrás situado en la carpeta donde está el fichero docker-compose.ymp. Para el despliegue, usa la orden:

```
docker compose up --scale cli=2 --scale wor=4
```

Importante: Dado que los nombres de imágenes son globales, al avanzar en esta práctica será conveniente indicar a "**docker compose**" que no emplee las imágenes de los anteriores apartados. Usa para ello alguna o varias de las órdenes de docker-compose (down, kill, rm, rmi)

Mientras se está ejecutando esta configuración, contesta las siguientes dos cuestiones:



Cuestión: Averigua la dirección IP de los 7 componentes desplegados (1 broker, 2 clientes y 4 trabajadores).

Cuestión: En la aplicación Visual Studio Code de las máquinas de portal se encuentra instalado el plugin para Docker. Observa toda la información que te puede proporcionar mientras se ejecutan los 7 componentes mencionados.

Comprueba que el funcionamiento es **correcto**...

- Cada cliente recibe respuesta a SU petición y no a las de otros
- No hay trabajadores libres si quedan peticiones pendientes
- Verificar (de nuevo) la tolerancia a fallos: hacer fallar un trabajador mientras atiende una petición y comprobar la reacción

Diseña modificaciones o formas de uso que faciliten las anteriores comprobaciones.

2 SESIÓN 2. ALMACENAMIENTO PERSISTENTE Y ACCESO REMOTO

Durante esta sesión, haremos 2 tareas.

1. **Depliegue CBWL**: Desplegaremos una nueva versión de la aplicación distribuida donde añadimos una componente adicional, el logger. La aplicación estará formada por un broker, clientes, trabajadores y la nueva componente logger.
2. **Cliente externo**: realizaremos accesos a nuestro despliegue mediante clientes externos.

PARTE 1. DESPLIEGUE CBWL

El código necesario para esta parte de la práctica está en la carpeta CBWL del material proporcionado.

2.1 Anotando los diagnósticos

Los componentes deben mostrar diagnósticos para notificar cómo progresan y si hay incidencias. Es frecuente que una buena parte de esas notificaciones empleen la salida estándar, pero es una práctica extendida que, salvo urgencia, esos diagnósticos se acumulen cronológicamente en algún archivo para su posterior consulta.

Podríamos elegir que cada componente guarde sus anotaciones en un fichero, pero no es cómodo contar con muchas fuentes de información. Si pretendiésemos que todos los componentes anotaran directamente sus diagnósticos en un único fichero centralizado, estaríamos rompiendo el concepto de sistema distribuido, donde cada componente está ubicada en un ordenador diferente, por tanto necesitamos otra aproximación.

La decisión que tomamos consiste en desarrollar un componente (***logger***) capaz de recibir mensajes de traza desde las demás componentes. Cuando queramos que una componente guarde una traza, la enviaremos mediante un mensaje a esta componente logger.

- **Por simplicidad, solo usaremos este servicio desde el componente broker,** pero es sencillo generalizarlo a todos los demás y sería lo que le daría un uso más completo.

Aspectos destacables:

- Es importante que el archivo usado por el logger no pierda su contenido entre invocaciones. Recuerda que la naturaleza efímera de los contenedores es aquí un problema a resolver. Utilizaremos un volumen Docker para conectar ese fichero con un espacio del anfitrión.
- Esta nueva componente logger deberá formar parte del despliegue y será una dependencia a resolver, pues las componentes que la utilicen deben conocer su ubicación.
- Un aspecto relevante es el tipo de socket ØMQ a utilizar. Nos decantamos por PULL para *logger* y PUSH para el resto (aunque otras variaciones permitirían otras características).

2.2 El componente logger y su efecto en el bróker

El logger se comunica con el resto de procesos actuando como un recolector, con patrón `PUSH-PULL`. Su código es muy simple. Además de la elección de sockets destaca la escritura en un fichero. **Edita el fichero logger.js y analiza su código.**

Por claridad, **únicamente el broker** hace uso del servicio de anotaciones, por lo que el código y el despliegue del broker deberán tenerlo en cuenta. Para ello hemos modificado el broker, al que ahora llamaremos **brokerl**. Las modificaciones han sido 3:

- Incorporar en la línea de órdenes dos nuevos argumentos (`loggerHost` y `loggerPort`)
- Un socket tipo PUSH para conectar con el componente logger

```
let slogger = zmq.socket('push')
conecta(slogger, loggerHost, loggerPort)
```

- Tras cada invocación a la función `traza`, añadir el envío del mismo texto (o algo similar) al logger

Edita el fichero brokerl.js e identifica estos cambios en el código.

El Dockerfile del logger es prácticamente idéntico a otros ya estudiados, destacando el argumento con **la ruta del directorio del contenedor** (¡¡no lo confundas con el del anfitrión!!)

El código completo de ambos componentes (brokerl y logger) así como las versiones actualizadas de los Dockerfile, están disponible en el material que habrás descargado para esta práctica, en la carpeta CBWL

2.3 Nuevas dependencias de los componentes

El brokerl necesitará conocer cómo conectar con *logger*. Esta situación es similar a la que ya relacionaba cliente y trabajador con el broker, y supone la necesidad de colocar una variable de entorno a sustituir en el despliegue. La última línea del Dockerfile del brokerl quedará:

```
CMD node mybroker 9998 9999 $LOGGER_HOST $LOGGER_PORT
```

2.4 Acceso a almacenamiento persistente desde logger

Es necesaria una consideración que no nos había preocupado hasta ahora: ¿cómo se relaciona el directorio con anotaciones (`/tmp/cbwlog`) del contenedor con el sistema de ficheros del anfitrión?. **Mediante una sección `volumes`[1] en la descripción del despliegue.**

Se supone que ya hemos creado el directorio `/tmp/logger.log` en el anfitrión. Comprueba que este directorio existe, o créalo si no existe.

Si únicamente deseáramos desplegar este componente logger, y no toda la aplicación distribuida, deberemos emplear una invocación de `docker run` con una opción equivalente a la sección `volumes`.

```
docker run -v /tmp/logger.log:/tmp/cbwlog parámetros
```

2.5 Despliegue conjunto del nuevo servicio CBWL

En el material de la carpeta CBWL dispones del nuevo `docker-compose.yml` que incluye el componente *logger*.

Despliega 4 clientes, 2 trabajadores, 1 broker y 1 logger.
sudo docker compose up --scale cli=4 --scale wor=2
Comprueba las anotaciones que ha efectuado el logger, accediendo en el anfitrión al directorio `/tmp/logger.log`. "broker starts"

Cuestión: Detalla los pasos necesarios para cambiar la ubicación del fichero de log a un nuevo directorio y pruébalo. We should change `/tmp/logger.log:/tmp/cbwlog` under `volumes:` in `docker-compose.yml` and rerun the docker compose command.
Cuestión: ¿Puedes modificar la función `traza` de `tsr.js`, empleada por el bróker, para incorporar el envío de mensajes al logger?

Cuestión: Reflexiona, sin necesidad de ejecutar, qué ocurriría si intentáramos desplegar los siguientes escenarios:

- 2 clientes, 1 trabajador, 2 brokers, 1 logger

[1] Dispones de información adicional en el material de referencia del tema 4

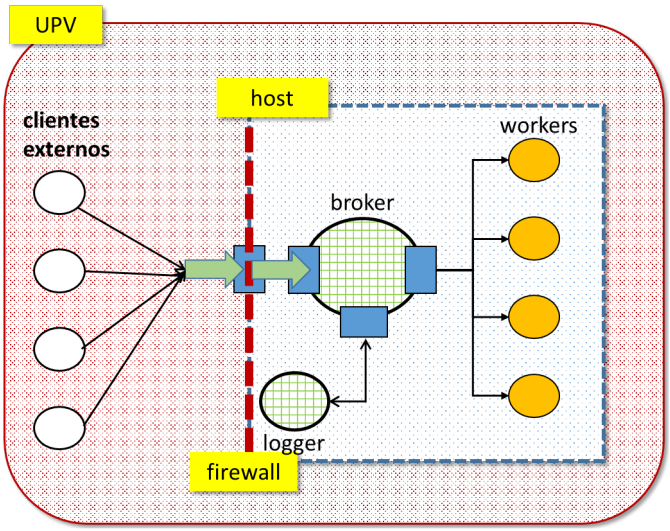
- 2 clientes, 1 trabajador, 1 broker, 2 loggers

Al final, para terminar los diferentes contenedores desplegados, ejecuta

```
docker-compose down
```

PARTE 2: ACCESO EXTERNO

En una visión realista, los clientes no son parte de la aplicación distribuida, y deberán interactuar con el servicio mediante algún punto bien conocido para poder encargarle trabajo. Un problema que aparece es la determinación del *endpoint* del servicio. Suponiendo que elijamos el URL del bróker:



- Si su IP puede cambiar en cada ejecución, no funcionará.
- Si los clientes se encuentran fuera del anfitrión que hospeda al resto de componentes, tendremos un problema de acceso (las IPs de Docker son locales dentro del anfitrión).

Ambos problemas pueden ser resueltos reservando un puerto del anfitrión que se hará corresponder con la IP y puertos del broker en el despliegue. Para ello hay que prestar atención a la línea "ports" de docker-compose.yml

- Si el cortafuegos del anfitrión (*host*) se encuentra configurado correctamente, la línea `ports` de `docker-compose.yml` hará el truco.
- Normalmente los equipos del portal en TSR suelen configurarse para permitir el acceso desde el exterior a los puertos 8000 a 9000. Para ampliar este soporte llegando al puerto 9999 ejecutamos:

```
sudo ufw allow 8000:9999/tcp
```

Ahora podremos disponer de clientes externos que conectarán con el servicio mediante un URL fijo `tcp://hostIP:9998` que conducirá las peticiones al broker.

Como cliente externo, necesitamos un ordenador con node+zeromq, más el código fuente del cliente. Este cliente debe ser capaz de "ver" la dirección IP o nombre de host completo de nuestro anfitrión. Esta dirección tendrá un aspecto como: `tsr-XXX.dsicv.upv.es` , donde `tsr-XXX` es el nombre de tu anfitrión.

Clientes válidos pueden ser las máquinas virtuales de otros compañeros, incluso la propia máquina anfitrion puede servir para hacer una primera comprobación.

Intenta hacer una prueba de funcionamiento. Para ello has de contemplar estos detalles:

1. Has adaptado la configuración de despliegue para añadir en la sección del broker:

```
ports:
  - "9998:9998"
```

2. Comprueba que puedes acceder al anfitrión desde el equipo remoto, mediante un simple "ping". (p.ej. con una orden `ping tsr-XXX.dsicv.upv.es`)

El código del cliente (en el directorio `cliente externo`) es idéntico al del cliente presentado en el sistema CBW, pero se ejecuta en el equipo de escritorio, lo que requiere que suministremos los argumentos necesarios.

3. Tomando los datos del ejemplo, deberíamos invocarlo de esta forma, para indicar un cliente con ID EXT que haga 2 invocaciones.

```
node cliente EXT 2 tsr-XXX.dsicv.upv.es 9998
```

4. Asegúrate de tener el servicio funcionando en el anfitrión.