

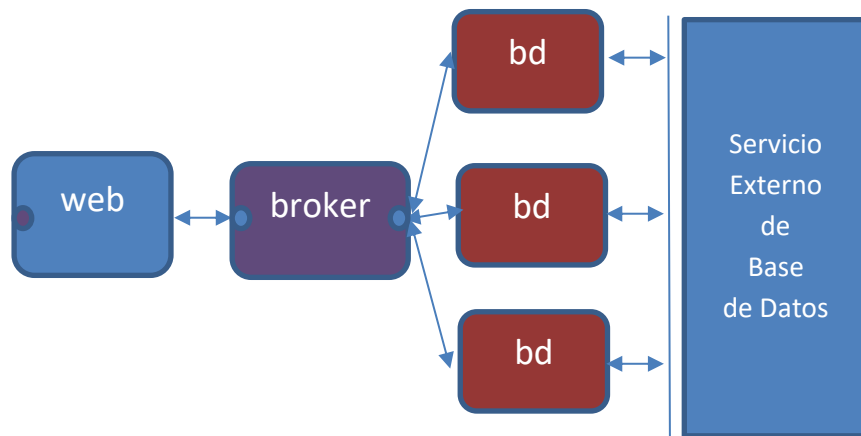
TSR: Actividades Tema 4 (Docker)

ACTIVIDAD 1. Test

1. Disponemos de un servicio multicomponente y pretendemos desplegarlo utilizando las herramientas de contenerización de Docker. El servicio consta de 3 componentes:

- **web**: recibe las peticiones por parte de los usuarios, a través de su anfitrión, y las envía a **broker**.
- **broker**: gestiona las peticiones y las reparte entre los diferentes componentes **bd** reenviando las respuestas a **web**.
- **bd**: accede a la información requerida por parte de los usuarios desde un servicio externo y devuelve al resultado a **broker**.

Deseamos realizar un despliegue de este servicio como el representado en la siguiente figura:



El componente **web** escucha en el puerto **8000**, accesible a través del puerto 80 de su anfitrión, mientras que el componente **broker** escucha en el puerto **8000** con su socket *frontend*, y en el puerto **8001** con su socket *backend*.

Indica cuál de los siguientes fragmentos del docker-compose.yml de este servicio es correcto:

a)	<pre>web: ... ports: - "80:8000" ... bd: ...</pre>	<pre>... broker: ... expose: - "8000" - "8001"</pre>
b)	<pre>web: ... expose: - "80" bd: ...</pre>	<pre>... broker: ... ports: - "8000" - "8001"</pre>

c)	<pre>web: ... expose: - "80:8000" bd: ...</pre>	<pre>... broker: ... expose: - "8000" - "8001"</pre>
d)	El servicio no puede lanzarse porque hay un conflicto de puertos entre el componente web y el componente broker , ya que ambos utilizan el puerto 8000 .	

2. Supongamos el ejemplo de la pregunta 1. Seguidamente se muestran los dockerfiles de los componentes **web**, **bd** y **broker** respectivamente (ambos utilizan node y zeromq, por lo que se pueden construir a partir de **tsr-zmq**):

```
FROM tsr-zmq
COPY ./web.js web.js
CMD node web $URL_BRO
```

```
FROM tsr-zmq
COPY ./bd.js bd.js
CMD node bd $URL_BRO
```

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

Si pretendemos realizar una correcta inyección de dependencias entre los componentes, indica cuál de los siguientes fragmentos del docker-compose.yml (versión 2) asociado al servicio, es correcto:

a)	<pre>web: ... links: - broker environment: - URL_BRO=tcp://broker:8000</pre>	<pre>bd: ... links: - broker environment: - URL_BRO=tcp://broker:8001 ... broker: ...</pre>
b)	<pre>web: ... links: - URL_BRO=tcp://broker:8000</pre>	<pre>bd: ... links: - URL_BRO=tcp://broker:8001 ... broker: ...</pre>

c)	<pre> web: ... links: - broker environment: - URL_BRO=tcp://bro:8000 </pre>	<pre> bd: ... links: - broker environment: - URL_BRO=tcp://bro:8001 ... broker: ... </pre>
d)	<pre> web: ... environment: - URL_BRO=tcp://broker:8000 bd: ... environment: - URL_BRO=tcp://broker:8001 ... </pre>	<pre> broker: ... links: - web - bd </pre>

3. Continuamos con el mismo ejemplo de las preguntas anteriores, donde cada directorio de componente incluye su código y su Dockerfile. Supongamos que...

- el directorio del servicio (donde se ha ubicado el docker-compose.yml) es “/home/service_web”
- el directorio del componente **web** es “/home/service_web/web”
- el directorio del componente **bd** es “/home/service_web/bd”
- y el directorio del componente **broker** es “/home/service_web/broker”.

Supongamos que ya está creada la imagen **tsr-zmq**. Indica cuál de los siguientes fragmentos del docker-compose.yml (junto con las anotaciones que los acompañan) es correcto si pretendemos lanzar el servicio con el comando “docker-compose up”:

a)	<pre> web: build: ./web ... bd: image: bd ... broker: build: ./broker ... </pre>	Antes de ejecutar el comando “docker-compose up” hemos de ejecutar el comando “docker build -t bd .” en el directorio donde está el fichero docker-compose.yml.
b)	<pre> web: image: ./web ... bd: image: ./bd ... broker: image: ./broker ... </pre>	
c)	<pre> web: image: web ... bd: build: ./bd ... broker: build: ./broker ... </pre>	Antes de ejecutar el comando “docker-compose up” hemos de ejecutar el comando “docker build -t web ./web” en el directorio donde está el fichero docker-compose.yml.
d)	<pre> web: build: ./web ... bd: build: ./bd ... broker: image: ./broker ... </pre>	Antes de ejecutar el comando “docker-compose up” hemos de ejecutar el comando “docker build -t broker ./broker” en el directorio donde está el fichero docker-compose.yml.

4. Para ese mismo sistema descrito en la pregunta 1, Indica cuál de las siguientes afirmaciones relacionadas con *docker-compose* es cierta:

a) Una orden para lanzar este servicio con 4 instancias del componente **bd** sería:

```
docker-compose up --scale bd=4
```

- b) La orden “**docker-compose stop**” elimina todos los contenedores en ejecución asociados al servicio lanzado con **docker-compose up**
- c) La orden “**docker-compose rm -f**” detiene temporalmente todos los contenedores en ejecución asociados al servicio lanzado con **docker-compose up**
- d) La única forma de eliminar los contenedores generados con **docker-compose up** es la utilización de la orden **docker rm -f <container_id>** para cada uno de esos contenedores.
5. Supongamos que un programador ha escrito un componente broker en NodeJS que escucha en múltiples puertos: 8000 (mensajes clientes), 8001 (conexión para modificar la configuración del broker) y 8002 (mensajes hacia los trabajadores). Cuando los clientes y los trabajadores se ubiquen en otros nodos, se recomienda asignar su puerto 8000 al puerto 80 del anfitrión y su puerto 8002 al 82 del anfitrión. El programa del broker se llama Broker.js y ese fichero está en la misma carpeta que este Dockerfile:

```
FROM tsr-zmq
COPY ./Broker.js /Broker.js
CMD node /Broker.js
```

Tras usar la orden “`docker build -t my-broker .`” en esa carpeta, se ha intentado ejecutar el broker, pero es incapaz de interactuar con algunos clientes y trabajadores remotos.

Para corregir ese problema, se debe:

a) Añadir esta línea en el Dockerfile:

```
PORT 8000 8001 8002
```

b) Añadir estas líneas en el Dockerfile:

```
PORT 80:8000 82:8002
PORT 8001
```

c) Añadir esta línea en el Dockerfile:

```
EXPOSE 8000 8001 8002
```

d) No se necesita hacer nada en el Dockerfile. El problema está relacionado con los argumentos y opciones utilizados en la orden “`docker run my-broker`”.

ACTIVIDAD 2. Publicador/Subscriptores

Este ejercicio gira en torno a la construcción de una aplicación distribuida basada en NodeJS+ZeroMQ 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('zeromq')
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)
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
  // Uncomment next line in order to stop when 100 messages have been sent.
  // 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
```

Dados 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:

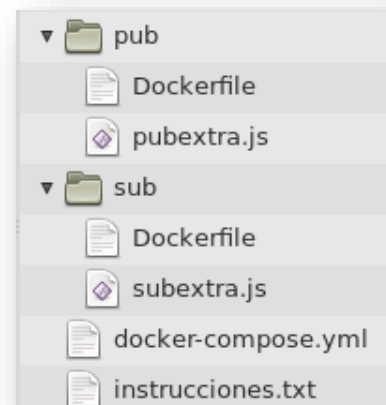
```
FROM tsr-zmq
...
```

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 que incluya a los dos mencionados.

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.

ACTIVIDAD 3. cbw ROUTER-ROUTER con tipos de trabajos

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

La aplicación distribuida con la que trabajaremos se encuentra descrita en el apartado 7.3 del documento `RefZMQ-xyz.pdf` utilizado en la práctica 2. Se recomienda repasar la descripción de esa variante concreta (**tipos de trabajos**) del patrón broker router/router.

En los aspectos que nos interesan, muy resumidamente, esta variante se caracteriza porque tanto el cliente como el trabajador incorporarán en su invocación un parámetro adicional `classID`.

Un ejemplo de invocación de `myworker1.js` es:

```
node myworker1.js localhost 8099 R
```

que, en este ejemplo, debería interpretarse como:

- Conecta con el *broker* en `tcp://localhost:8099`
- Admite procesar peticiones de tipo `R`

En el caso de los clientes (`myclient1.js`), su invocación podría ser:

```
node myclient1.js localhost 8098 G
```

que, en este ejemplo, debería interpretarse como:

- Conecta con el *broker* en `tcp://localhost:8098`

- Anuncia que la petición de servicio es del tipo G

Suponemos que en el anfitrión se encuentra disponible una imagen Docker inicial denominada **tsr-zmq** que incluye todo el software necesario para ejecutar los procesos anteriores, pero cada componente deberá añadir sus requisitos específicos para ser desplegado.

- Los clientes siempre serán del tipo **B** (dato desconocido para el resto).
- Se dispondrá de *workers* para cada uno de los 3 tipos (dato desconocido para el resto).
- Por simplicidad, no nos preocupamos de la biblioteca `tsr.js` de la práctica 2

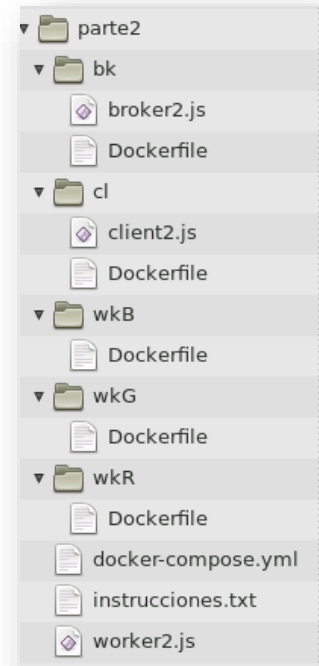
Debe desarrollarse el `Dockerfile` de cada uno, el `docker-compose.yml` que relacione estos componentes, y ejecutar el escalado que más tarde se propone.

Para completar este ejercicio debe crearse una carpeta `parte2`, ilustrada a la derecha, con un directorio `parte2/bk` para el material relacionado con el *broker*, otro denominado `parte2/cl` para el material necesario en el despliegue de los clientes, mientras que el archivo `docker-compose.yml` debe encontrarse directamente en `parte2`. En el caso de los trabajadores colocaremos un directorio `parte2/wkB` que contenga todo el material para el despliegue de los workers de tipo B, `parte2/wkG` para los que atienden peticiones de tipo G, y `parte2/wkR` para los de tipo R.

- Observarás que se especifica un único `worker2.js` para que su código sea compartido por todos los `wk{RGB}`.

Junto con esta estructura el alumno debe elaborar (escribir) un archivo `instrucciones.txt` que indique con exactitud:

1. La secuencia de órdenes necesarias para construir los componentes.
2. Las instrucciones necesarias para desplegar, tal y como ya se ha dicho, una aplicación distribuida compuesta por:
 - 2 clientes de tipo B,
 - 3 workers (1 de cada tipo),
 - y 1 broker.
3. Las instrucciones necesarias para escalar a 6 clientes de tipo B, 6 workers (2 de cada tipo) y 1 broker.



ACTIVIDAD 4. Secuenciador para ordenar acceso a variables

Observa estos dos programas Proc.js y Seq.js

```
// Proc.js
var zmq = require ('zmq')
if (process.argv.length!=4) {
  console.error('Usage: node proc seqIP procID');
  process.exit(1);
}
var local = {x:0, y:0, z:0}
var port = {x:9997, y:9998, z:9999}
var ws = zmq.socket ('push')
ws.connect('tcp://'+process.argv[2]+':8888')
var rs = zmq.socket ('sub')
rs.subscribe('')
for (var i in port)
  rs.connect('tcp://'+process.argv[2]+'-'+port[i])
var id = process.argv[3]

function W( name, value ) {
  console.log("W"+ id +"(" + name + ")" + value )
}
function R( name ) {
  console.log("R"+ id +"(" + name + ")" + local [ name ])
}

var n=0, names=["x","y","z"]
function writeValue() {
  n ++; ws.send ([names[n%names.length], (10*id)+n, id])
}
rs.on('message', function(name, value, writer) {
  local[name] = value;
  if (writer==id) W(name, value); else R(name)
})
function work() { setInterval( writeValue, 10) }
setTimeout(work, 2000); setTimeout(process.exit, 2500)
```

```
// Seq.js
const zmq = require ('zmq')
var port = { x:9997, y:9998, z:9999}
var s = {}

var pull = zmq.socket ("pull")
pull.bindSync('tcp://*:8888')
for (var i in port ) {
  s[i]=zmq.socket('pub')
  s[i].bindSync('tcp://*:'+port[i])
}
pull.on('message',
  function( name, value, writer ) {
    s[name].send([name, value, writer])
  })
```

Estos programas implantan el modelo de consistencia “caché”. Proc.js emula un proceso que comparte tres variables: “x”, “y” y “z”. Seq.js es un secuenciador que garantiza un orden común para los valores de cada una de las variables compartidas.

Queremos desplegar tres procesos Proc.js con identificadores (procID) 1, 2 y 3 y un proceso “Seq.js”.

Responde las siguientes cuestiones relacionadas con ese despliegue. Para ello, asume que en el anfitrión hay una imagen Docker local basada en “ubuntu:focal” con las órdenes “node” y “npm”, la biblioteca ZeroMQ y el módulo NodeJS “zmq” instalados correctamente. El nombre de esa imagen es “tsr-zmq”:

1. Escribe un Dockerfile para desplegar el componente “Seq.js”. Asume que ese Dockerfile estará en la misma carpeta donde resida “Seq.js”.

2. Escribe la orden para generar una imagen llamada “seq” con ese Dockerfile.
3. Escribe la orden para ejecutar un contenedor que use esa imagen “seq”.
4. Averigua la dirección IP que se ha asignado al contenedor del secuenciador. Escribe un Dockerfile para desplegar el componente “Proc.js” y que pueda interactuar con el componente “seq”.
5. Prepara el despliegue de un servicio compuesto por 1 componente “seq” y 3 componentes “proc”, conociendo que cada “proc” debe recibir un identificador procID con valor 1, 2 ó 3 (todos diferentes).
6. Explica qué cambios necesitaremos en los apartados anteriores (idealmente solo en los que no dependen de “seq”) para desplegar cada uno de esos componentes (seq, proc 1, proc 2 y proc 3) en un anfitrión diferente. Para ello, asume que la dirección IP del anfitrión donde se ejecuta el secuenciador es 192.168.0.10.