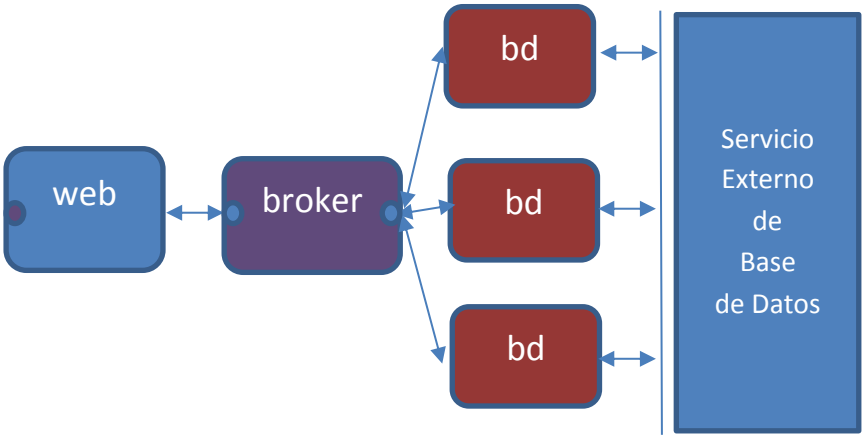
TSR: Actividades Tema 4 (Docker)

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)

web: … ports: -“80:8000” … bd: …	… broker: … expose: -“8000” -“8001”

b)

web: … expose: -“80” bd: …	… broker: … ports: -“8000” -“8001”

c)

web: … expose: -“80:8000” bd: …	… broker: … expose: -“8000” -“8001”

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 zmq, por lo que se pueden construir a partir de **centos-node-devel**)**:**

```
FROM centos-node-devel
COPY ./web.js  web.js
CMD node web $URL_BRO
```

```
FROM centos-node-devel
COPY ./bd.js  bd.js
CMD node bd $URL_BRO
```

```
FROM centos-node-devel
COPY ./broker.js  broker.js
CMD node broker 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)

web: … links: - broker environment: - URL_BRO=tcp://broker:8000	bd: … links: - broker environment: - URL_BRO=tcp://broker:8001 … broker: …

b)

web: … links: - URL_BRO=tcp://broker:8000	bd: … links: - URL_BRO=tcp://broker:8001 … broker: …

c)

web: … links: - broker environment: - URL_BRO=tcp://bro:8000	bd: … links: - broker environment: - URL_BRO=tcp://bro:8001 … broker: …

d)

web: … environment: - URL_BRO=tcp://broker:8000 bd: … environment: - URL_BRO=tcp://broker:8001 …	broker: … links: - web - bd

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 **centos-node-devel**. 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)

web: build: ./web … bd: image: bd … broker: build: ./broker …	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)

web: image: ./web … bd: image: ./bd … broker: image: ./broker …	

c)

web: image: web	Antes de ejecutar el comando "docker-compose up**"**

	… bd: build: ./bd … broker: build: ./broker …	hemos de ejecutar el comando **"**docker build –t web ./web**"** en el directorio donde está el fichero docker-compose.yml**.**
d)	web: build: ./web … bd: build: ./bd … broker: image: ./broker …	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 secuencia de órdenes para lanzar este servicio con 4 instancias del componente **bd** sería:

```
docker-compose up
docker-compose 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 zmq-devel
COPY ./Broker.js /Broker.js
CMD node /Broker.js
```

Tras usar la orden "docker build –t myBroker ." en esa carpeta, se ha intentado ejecutar el broker, pero es incapaz de interactuar con algunos clientes y trabajadores locales.

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 myBroker”.

ACTIVIDAD 2. Publicador/Subscriptores

Este ejercicio gira en torno a la construcción de 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
```

Dados ya la descripción y código de los componentes, se desea desplegarlos en contenedores de nuestra máquina virtual, constituyendo una aplicación distribuida.

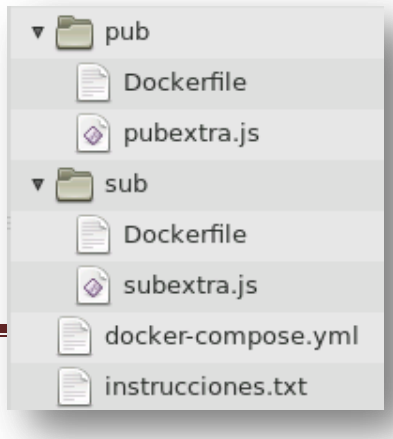
Para ello deberás elaborar un fichero de configuración Dockerfile para el publicador, y otro compartido por los suscriptores.

Los requisitos básicos *incluyen* estas instrucciones:

```
FROM centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_8.x | bash -
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
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 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 trabajo

Este ejercicio gira en torno a la modificación del despliegue de una aplicación distribuida basada en NodeJS+ZMQ 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-cas.pdf utilizado en la práctica 2. Se recomienda repasar la descripción de esa variante del esquema "intermediario entre clientes y trabajadores".
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 soyWorker1 localhost:8099 R
```

que, en este ejemplo, debería interpretarse como:

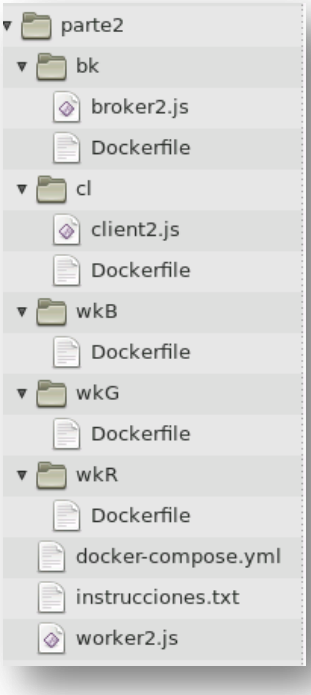
- El identificador de este worker es soyWorker1
- Conecta con el broker en tcp://**localhost**:**8099**
- Admite procesar peticiones de tipo R

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

```
node myclient1.js soyClient1 localhost:8098 G
```

que, en este ejemplo, debería interpretarse como:

- El identificador de este cliente es soyClient1
- 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 tsr1718/centos-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).

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 "centos:latest" con las órdenes "node" y "npm", la biblioteca ZeroMQ y el módulo NodeJS "zmq" instalados correctamente. El nombre de esa imagen es "**exercise04**":

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. Asumamos que ese secuenciador está ejecutándose en un contenedor cuya dirección IP es 172.17.0.3. Escribe un Dockerfile para desplegar el componente “Proc.js” y que pueda interactuar con el componente “seq”.
5. Explica qué cambios necesitaremos en los apartados anteriores (idealmente solo en el 3 y el 4) 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.

Solución 1. Test

1. a)
2. a)
3. c)
4. a)
5. d)

Solución 2. Publicador/Subscriptores

instrucciones.txt

Suponiendo que pubextra.js se encuentra en pub, que subextra.js se encuentra en sub, y que nuestro directorio actual es el que incluye a esos dos.

```
#!/bin/sh
for i in pub sub
do
  cd $i
  docker build  -t ${i}extra .
  cd ..
done
docker-compose up --scale 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 centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_8.x | bash -
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
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 centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_8.x | bash -
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN npm install zmq
COPY ./subextra.js subextra.js
CMD node subextra.js $PUB_URL ocio
```

Solución 3. Esquema cbw ROUTER-ROUTER con tipos de trabajo

Solución 4. Secuenciador para ordenar acceso a variables

1. Para escribir el Dockerfile debemos estudiar el programa "Seq.js". Utiliza un único socket PULL para recibir mensajes del resto de procesos. Ese socket está ligado al puerto 8888. Además, emplea un socket PUB para cada variable, de manera que pueda difundir sus escrituras cuando hayan sido secuenciadas. Esos sockets PUB se asocian también a puertos locales. Dado que se dispone de tres variables diferentes ("x", "y" y "z"), se necesitarán tres puertos: 9997, 9998 y 9999.

Por tanto, hay que desarrollar un Dockerfile que "exponga" esos cuatro puertos. Puede parecerse a otros que se han estudiado con anterioridad. También hemos de considerar que la imagen de partida es "exercise04". Con todas estas consideraciones, el resultado puede ser:

```
FROM exercise04
COPY ./Seq.js /
EXPOSE 8888 9997 9998 9999
CMD node Seq
```

2. Si asumimos que en nuestro directorio actual se encuentra ese Dockerfile, la orden empleada para generar esa imagen es:

```
docker build –t seq .
```

3. Para poner en marcha el contenedor, usaremos la siguiente orden:

```
docker run seq
```

4. Aunque solo hay un componente "proc", se dispondrá de tres instancias con diferentes procIDs. Por tanto, en el momento de escribir el Dockerfile que genera la imagen "proc" encontramos un problema: desconocemos el valor adecuado para los argumentos que se necesitan para iniciar los procesos "proc". Si esa información apareciera especificada (estáticamente) en el Dockerfile, necesitaríamos modificar ese fichero y reconstruir la imagen para cada instancia a ejecutar.

Realicemos una primera aproximación con esa versión estática. Sus contenidos serán:

```
FROM exercise04
COPY ./Proc.js /
CMD node Proc 172.17.0.3 1
```

Pero esto significa que, para ejecutar los tres procesos mencionados en el enunciado, deberíamos modificar la última línea del fichero, sustituyendo su último argumento (1) con otros valores (2 y 3, respectivamente) de manera que se generen tres imágenes "proc" diferentes, una por cada proceso a emular.

Esta solución, pese a ser válida, exige demasiado esfuerzo del usuario.

Este ejercicio sugiere otra alternativa: escribir un único Dockerfile para generar una única imagen compartida por los tres procesos. Para ello hemos de sustituir la última línea del Dockerfile por otras dos que se complementan: una con la orden ENTRYPOINT, indicando el programa a ejecutar, y una segunda con la orden CMD, especificando los argumentos a usar por omisión. Tales argumentos pueden ser reemplazados fácilmente por otros valores mediante una orden "docker run".

El nuevo Dockerfile que obtenemos es:

```
FROM exercise04
COPY ./Proc.js /
ENTRYPOINT ["node","Proc"]
CMD ["172.17.0.3","1"]
```

Tanto ENTRYPOINT como CMD usan la sintaxis basada en vector para recibir sus argumentos. La razón que nos obliga es evitar que el *shell* intervenga en el procesamiento de estos valores. Para generar la imagen "proc", asumiendo que en nuestro directorio actual se encuentra el Dockerfile anterior, la orden será:

```
docker build –t proc .
```

Con todo lo anterior, podemos arrancar las tres instancias de "proc" con las órdenes:

```
docker run proc
docker run proc 172.17.0.3 2
docker run proc 172.17.0.3 3
```

5. Los cambios necesarios serán:
    - En el momento de lanzar el componente "seq", tal como se solicita en el apartado 3 del ejercicio, debemos asociar los puertos "expuestos" con los puertos reales del anfitrión. Por ejemplo, podemos especificar los mismos puertos en anfitrión y contenedor.

La opción necesaria en "docker run" es "-p". Un ejemplo de este tipo es:

```
docker run –p 8888:8888 –p 9997-9999:9997-9999 seq
```

- El otro cambio consiste en sustituir la dirección IP especificada en el Dockerfile de "proc" (p.ej. 172.17.0.3, la dirección del contenedor de "seq") con la del anfitrión de "seq" (p.ej. 192.168.0.10 en este ejemplo). Con esto ya se da respuesta a esta parte de la pregunta.

Para desarrollar una respuesta más completa, podemos extenderla de la forma que se describe a continuación.

- En la anterior solución "manual" nos veíamos obligados a regenerar las tres imágenes empleadas por los tres procesos, colocando en su última línea la dirección IP apropiada.
- Por otro lado, con nuestra imagen "proc" mejorada solo necesitamos cambios menores en las órdenes "docker run" que deben emplearse para iniciar esos procesos. En el último caso, las nuevas órdenes a usar son:

```
docker run proc 192.168.0.10 1
docker run proc 192.168.0.10 2
docker run proc 192.168.0.10 3
```

Y cada orden deberá invocarse en el equipo donde vaya a ejecutarse cada instancia.