TSR – 9 de enero de 2019. EJERCICIO

Considérense estos dos programas que implantan un servicio de “chat”:

```
// Client.js
const zmq=require('zmq')
const ps=zmq.socket('push')
const sub=zmq.socket('sub')

ps.connect('tcp://127.0.0.1:8000')
sub.connect('tcp://127.0.0.1:8001')

sub.subscribe('')
sub.on('message', (msg) => {
        console.log(msg+'')
})

let userid=process.argv[2] || 'user'+process.pid

let conMsg={type:'connect',user:userid}
ps.send(JSON.stringify(conMsg))

process.stdin.resume()
process.stdin.setEncoding('utf-8')
process.stdin.on( 'data', (line) => {
        let chatMsg = {type:'chat', user:userid, data:line}
        ps.send(JSON.stringify(chatMsg))
})

function disconnect() {
        let disMsg = {type:'bye', user:userid}
        ps.send(JSON.stringify(disMsg))
        setTimeout( process.exit,500)
}

process.on('SIGINT', disconnect )
process.stdin.on('end', disconnect )
```

```
// Server.js
const zmq=require('zmq')
const pull=zmq.socket('pull')
const pub=zmq.socket('pub')

pull.bindSync('tcp://*:8000')
pub.bindSync('tcp://*:8001')

pull.on('message', (msg) => {
   let m=JSON.parse(msg)
   let answer=''
   switch(m.type) {
      case 'connect':
         answer='Server> chat-client '+m.user+' connected!'
         break
      case 'chat':
         answer=m.user+'> '+m.data
         break
      case 'bye':
         answer='Server> chat-client '+m.user+' disconnected!'
   }
   pub.send(answer)
})
```

Responda a estas cuestiones relacionadas con el despliegue de esos elementos con Docker. Para ello, asuma que en el ordenador anfitrión existe una imagen Docker basada en “centos:latest” con las órdenes **node** y **npm**, la biblioteca ZeroMQ y el módulo NodeJS **zmq** adecuadamente instalados. El nombre de esa imagen es “**exercise02**”:

1. Revise el código de ambos programas, concretamente las URL usadas en las operaciones bindSync() y connect() y describa si se necesita algún cambio para su despliegue en contenedores. De ser así, escriba y explique el fragmento de programa necesario para gestionar todo aquello relacionado con bind y connect. (2 puntos)
2. Escriba un Dockerfile para desplegar el componente “Server.js”. Asuma que ese Dockerfile está en el mismo directorio que “Server.js”. (2 puntos)
3. Escriba la orden necesaria para construir una imagen llamada “server” con ese Dockerfile. (1 punto)
4. Escriba la orden para ejecutar un contenedor que use la imagen “server”. (1 punto)
5. Asuma que ese servidor se ejecuta en un contenedor cuya dirección IP es 172.17.0.3. Escriba un Dockerfile para desplegar el componente “Client.js”. Debe ser capaz de interactuar con el servidor en ejecución. (2 puntos)
6. Escriba y explique las órdenes necesarias para ejecutar un contenedor con la imagen generada a partir del Dockerfile del punto anterior. Preste atención a las opciones “-i” y “-t” (1 punto).
7. **Explique** qué acciones deberíamos aplicar en los apartados anteriores para desplegar cada uno de esos componentes (un servidor y varios clientes) en diferentes anfitriones. Para ello, asuma que la dirección IP del anfitrión del servidor es la 192.168.0.10. (1 punto)

NOTA: En este ejercicio no se necesita **docker-compose**. Todos los apartados pueden resolverse utilizando acciones de la orden **docker** (p.ej., **build**, **run**, **commit**, **ps**, **images**, **rm**, **rmi**...)

SOLUCIÓN

1. Ambos programas suponen que se ejecutarán en el mismo ordenador. Por eso el servidor emplea el símbolo "*" para realizar un bind a todas las direcciones IP locales, y el cliente emplea 127.0.0.1 como dirección IP local (interfaz *loopback*). Si hay que ejecutar esos programas en contenedores, se necesitará adaptar el código del cliente: debe conectar a la dirección IP del servidor cuando éste (el servidor) se ponga en marcha. Para ello el cliente necesita recibir ese valor (la dirección IP del servidor) como argumento de la línea de órdenes. La versión actual del código del cliente ya supone la existencia de un argumento con el nombre (userid) del cliente. Por tanto modificaremos la lista de argumentos, colocando la dirección del servidor en primera posición, y desplazando el nombre opcional de usuario a una segunda posición.

   Como resultado de todo lo anterior, las primeras líneas del cliente quedarán así:

```
// Client.js
const zmq=require('zmq')
const ps=zmq.socket('push')
const sub=zmq.socket('sub')

// Check how many arguments have been received.
if (process.argv.length < 3) {
        console.log('These arguments are needed:')
        console.log('  -  Server IP address.')
        console.log('  -  User nickname.')
        // Abort if no argument has been given.
        process.exit(1)
}
// Get the server IP address from the appropriate command-line argument
const addr = process.argv[2]

// Build the server URLs and connect to them.
ps.connect('tcp://'+addr+':8000')
sub.connect('tcp:// '+addr+':8001')

sub.subscribe('')
sub.on('message', (msg) => {
            console.log(msg+'')
})

let userid=process.argv[3] || 'user'+process.pid
...
```

2. El contenido del Dockerfile será:

```
FROM exercise02
COPY Server.js /
EXPOSE 8000 8001
CMD node Server
```

3. La orden necesaria para construir la imagen solicitada es:

```
docker build –t server .
```

   Observa que esta orden debe ejecutarse dentro de la carpeta que contenga el Dockerfile de la parte 2.

4. Orden para ejecutar el servidor dentro de un contenedor:

```
docker run server
```

   Observa que esta orden puede ejecutarse en cualquier carpeta del anfitrión cuyo depósito docker contenga la imagen "server".

5. El Dockerfile básico que se ajusta a los requisitos de esta parte es:

```
FROM exercise02
COPY Client.js /
CMD node Client 172.17.0.3
```

Observa que, en esta primera solución, hemos asumido que el servidor siempre se ejecuta en un contenedor cuya dirección IP es 172.17.0.3. Sin embargo, éste no es un supuesto realista (**pero en este examen no se solicita ninguna de las extensiones que se comentan a continuación**). Ya que sabemos que en cada despliegue puede cambiar la dirección IP del servidor, podemos proponer una segunda versión adaptativa:

```
FROM exercise02
COPY Client.js /
CMD node Client $SERVER_IP
```

Si sabemos combinar las instrucciones ENTRYPOINT y CMD, podemos incluso mejorar la solución con esta tercera versión:

```
FROM exercise02
COPY Client.js /
ENTRYPOINT ["node","Client"]
CMD ["172.17.0.3"]
```

6. Para poder ejecutar un cliente se necesitan varios pasos. En primer lugar, hay que generar una imagen docker usando el Dockerfile del apartado 5 (eligiendo entre la versión básica o cualquiera de sus mejoras). Para ello, en la misma carpeta en que se encuentre ese Dockerfile, ejecutamos la orden:

```
docker build –t client .
```

Como resultado, en el depósito docker local se creará una nueva imagen "client". Ahora necesitamos ejecutar al menos una instancia de dicha imagen. Si se trata de la versión básica, la orden necesaria para ejecutar un cliente será:

```
docker run –i –t client
```

Alternativamente, si empleamos alguna de las extensiones propuestas (**no se necesitan para que la solución se considere válida**), las consideraciones son:

- Para la segunda versión se necesita una orden como:

```
docker run –i –t –e SERVER_IP=172.17.0.3 client
```

- Para la tercera versión puede emplearse la misma orden que en el caso básico, pero, además, podemos cambiar el nombre (userid) suministrado a cada instancia del cliente:

```
docker run –i –t client 172.17.0.3 myName
```

En este caso el usuario puede emplear argumentos de la línea de órdenes que sustituirán a los argumentos por omisión indicados en la orden CMD del Dockerfile del cliente.

Para poder suministrar un nombre de cliente (p.ej. "myOtherNickname") en el resto de versiones (básica y segunda), debemos incluir una línea de órdenes completa tras el nombre de la imagen, de forma que sustituyan los valores por omisión indicados en la instrucción CMD del Dockerfile del cliente; algo así como …

```
docker run –i –t client node Client 172.17.0.3 myOtherNickname
```

7. Los cambios aplicados en las partes anteriores deben concentrarse en la correspondencia (*mapping*) entre los puertos del contenedor que ejecuta Server.js y los de su anfitrión. Además, los clientes deben usar ahora la dirección IP del anfitrión en el que se ejecuta el contenedor, en vez de la IP del propio contenedor. Como consecuencia, considerando la versión básica de la solución, deben aplicarse los dos cambios siguientes:
   - En el apartado 4, los puertos del contenedor para el servidor se vinculan con los del mismo número de su anfitrión mediante la orden:

     ```
     docker run –p 8000:8000 –p 8001:8001 server
     ```

   - En el apartado 5, debemos sustituir la dirección 172.17.0.3 al final del Dockerfile con la dirección IP del anfitrión, p.ej., 192.168.0.10. Tras aplicar el cambio, debería volver a repetirse todo el proceso de construcción y ejecución de la imagen cliente resultante, siguiendo los pasos descritos en el apartado 6, sin necesidad de modificación en caso de emplear la solución más básica.