TSR – 18th January 2018. EXERCISE 4

Given these two programs:

```
// 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])
        })
```

Those two programs provide an implementation of the *cache* consistency model. "Proc.js" provides an emulation for processes that share three variables: x, y and z. "Seq.js" is a sequencer that is used in this implementation for ensuring a common order of values on each of the shared variables.

We are trying to deploy three "proc.js" processes with identifiers (procID) 1, 2 and 3 and a "Seq.js" process.

Please, answer the following questions related to that deployment. To this end, let us assume that the host computer has a local Docker image based on "centos:latest" with the **node** and **npm** commands, the ZeroMQ library, and the **zmq** NodeJS module properly installed. The name of that image is "**exercise04**":

1. Write a single Dockerfile for deploying the "Seq.js" component. Assume that such a Dockerfile is in the same folder where the file "Seq.js" is placed (3 points).
2. Write a command for generating an image called "seq" using that Dockerfile (1 point).
3. Write a command to execute a container that uses the "seq" image (1 point).
4. Let us assume that such sequencer is running in a container whose IP address is 172.17.0.3. Write a single Dockerfile for deploying the "Proc.js" component, matching the seq component (3 points).
5. **Discuss** which changes are needed in some of the previous parts (ideally, only in parts 3 and 4) in order to deploy each one of those components (seq, proc 1, proc 2 and proc 3) in different hosts. To this end, let us assume that the IP address of the host where the sequencer is running is 192.168.0.10 (2 points).

SOLUTION

1. In order to write this Dockerfile we should revise the program "Seq.js". It uses a single PULL socket for receiving the messages from the other processes. That socket is bound to port 8888. Besides, it uses as many PUB sockets as variables exist, in order to multicast their writes once they have been sequenced. Those PUB sockets are also bound to local ports. Since there are three different variables ("x", "y" and "z"), three ports are needed to this end: 9997, 9998 and 9999.
   Thus, we need a Dockerfile that "exposes" those four ports. It may be similar to others we have previously written. We should also consider that its base image is "exercise04". The result may be similar to this:

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

2. The command to be used in order to generate that image is the following one, assuming that this command is run in the Dockerfile's folder:

```
docker build –t seq .
```

3. The command to be used in order to run that container is the following one:

```
docker run seq
```

4. Although there is a single "proc" component, it will have three instances with different procIDs. Therefore, we have a trouble at the time of writing the Dockerfile for building the "proc" image, since those arguments should be passed to the resulting "proc" processes when they are started. If such information is statically set in the Dockerfile, we will need to modify the Dockerfile and rebuild the image per each process to be run.
   Let us start with that "uncomfortable" static version. Its contents may be the following:

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

   But this means that, in order to run the three processes mentioned in the wording, we should modify the last line in the file, replacing its last argument (1) with other values (2 and 3, respectively) in order to generate three different "proc" images, one per process to be emulated.
   Such a solution is considered valid, but it requires too much effort from the user.
   This exercise is suggesting another alternative: to write a single Dockerfile for generating a single image that will be shared by all processes to be run. To this end, we need to replace the last line of the Dockerfile with two complementary lines: one with the ENTRYPOINT keyword, stating the program to run, and a second one with the CMD keyword, stating the default arguments to use. Those arguments may be easily replaced passing other values as final arguments in the "docker run" command.
   The resulting "evolved" Dockerfile is:

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

   Both ENTRYPOINT and CMD are using their array-based syntax for receiving their arguments. This is needed for avoiding any shell intervention in the processing of those arguments.
   The "proc" image is generated using this command in the same folder where this Dockefile is placed:

```
docker build –t proc .
```

   With this, the three "proc" instances may be started using the following commands:

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

5. The wording asks for a discussion. The explanation to be given by each student does not need to provide the concrete commands being needed for achieving that functionality, since no Docker documentation is available at the time of answering this question.
   Thus, the changes to be applied are the following:

- At the time of launching the “seq” component, as requested in part 3 of the exercise, we must map its “exposed” ports to real ports in its host. For instance, we may use in the host the same port numbers than in the container.

  The “docker run” option to be used is the “-p” one. Thus, an example is:

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

- The other change consists in replacing the IP address being assumed in the “proc” Dockerfile (i.e., 172.17.0.3, that of the “seq” container) with that of the “seq” host (i.e., 192.168.0.10 in this example). This already provides the required answer in this part of the question.
  In order to write a more complete answer, it can be extended as follows.
  In our “manual” solution described above, we are compelled to regenerate the three images used by the three emulated processes, placing in their last line the appropriate IP address value.
  On the other hand, with our “evolved” “proc” image, we only need small changes in the “docker run” commands to be used for starting those processes. In the latter case, the new commands to be used are:

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

  And each one of those commands will be used in the computer where each instance is going to run.