TSR / NIST – Lab 3 (Retake)

This exam consists of 5 multiple choice questions. In every case only one answer is correct. You should answer in a separate sheet. If correctly answered, they contribute 2 points to the exam grade. If incorrectly answered, the contribution is negative: -0.667. So, think carefully your answers.

1. Let us assume a Docker image called tsr1718/centos-nodejs. The Docker command to be used for generating a Docker container based on that image is:

X	docker run tsr1718/centos-nodejs Yes. With the "docker run" command we may start a container. To this end we should specify the Docker image name or ID, as we have done in this example.
	docker build . No. See the explanation in the next alternative.
	docker build -t tsr1718/centos-nodejs No. The "docker build" command generates a Docker image using a Dockerfile to this end. Besides, it needs as a compulsory argument the path of the folder where such Dockerfile is placed. That argument is missing in this example.
	docker-compose up No. The "docker-compose up" command may start multiple containers. The names of the Docker images to be used to this end are specified in a "docker-compose.yml" file.

2. In order to deploy the TCPProxy component without using the dependency injection mechanisms from the docker-compose command, we must...

X	...find out the IP address using “docker inspect” on the web server container. True. The TCPProxy depends on the endpoint of the web server. Therefore, we need to know the IP address of that web server. When the web server is run in a container, we may use "docker inspect" in order to find its IP address.
	...find out the needed address using “docker inspect” on the TCPProxy image. No. In order to solve the dependences of the TCPProxy component, we cannot look for a datum that belongs to the TCPProxy itself. Actual dependences are set on the endpoints of the other components.
	...find out the IP address of the TCPProxy container using “ifconfig”. No. In order to deploy a TCPProxy component, we cannot depend on a datum that can only be known once the TCPProxy itself is running. That is a nonsense.
	No IP address needs to be found out, since Docker does this automatically. False. See the explanation in the correct part.

3. When the TCPProxy is connected to the minimal web server (without using the “docker-compose” command)...

X	In order to run the minimal web server, we DO NOT NEED any “-p 8000:80” modifier in the “docker run” command. True. That modifier is needed for mapping port 80 in the container to port 8000 in the host. Since both the web server and the TCPProxy may run in the same host, there is no need for that mapping.

	In the web server Dockerfile no “EXPOSE 80” instruction is needed. False. A Dockerfile states the actions to be considered in the creation of an image. Its contents do not depend on whether the "docker-compose" command will be used afterwards or not. The "EXPOSE 80" instruction is needed for reporting that this web server listens for incoming connections in port 80. That instruction cannot be removed.
	We cannot connect those two components without using the “docker-compose” command. False. They can be run without trouble when the "docker-compose" command is not used. To this end: 1) The web server is started first using "docker run". 2) With "docker inspect" we may find out the IP address of that container. 3) Such IP address is passed to the TCPProxy component either updating its Dockerfile (in case of writing there the list of arguments to be passed to that proxy) and rebuilding its image or passing that information as additional arguments in the "docker run" command (when the information to be received by the proxy is specified using environment variables in its Dockerfile or as CMD-based arguments).
	In order to build the URL to be used by the browser, we should use the IP address of the virtual desktop where we are working. No. It is worth noting that the virtual desktop does not behave as a host for any of those components since they use Docker on the virtual machines.

4. Regarding the inclusion of a “logger” component in the cbw (client-broker-worker) application...

X	The “logger.log” log file should already exist when the distributed application execution is started. True. The “logger” component writes its information in that file. The file must exist in order to properly append information onto it.
	The needed updates do not affect the other components: client, worker and broker. False. When the “logger” is added all other components need to report their trace messages to it. To this end, they need another ZeroMQ socket and their components need a “link:” block in the “docker-compose.yml” file.
	In order to communicate with the “logger”, each remaining component reconnects any of the ZeroMQ sockets that it already uses. False. Since the “logger” component uses a PULL socket, all the other elements need to add a PUSH socket in order to communicate with it. They cannot reuse their previous sockets.
	The “logger” component is not included in the “docker-compose.yml” file since its specification is in its own Dockerfile. No. It is included in the “docker-compose.yml” file. Besides, it also has its own Dockerfile.

TSR / NIST – Lab 3 (Retake)

5. Given two cbw (client-broker-worker) systems A and B, as shown in lab 3. When those two systems are chained, this can be achieved...

X	Some workers in A may behave as clients in B. True. That situation introduces service/system chaining and it has been explained in this lab using the "worcli" component.
	The broker in A may behave as a broker in B. No. In that case there is no chaining. Besides, this scenario basically corresponds to handling A and B as two job classes in a single service. That situation was already seen in lab 2 and it does not introduce service chaining.
	Some clients in A may behave as clients in B. No. In that case there is no chaining between those systems.
	Some workers in A may behave as workers in B. No. In that case there is no chaining in those systems.