TSR - LAB 3
COURSE 2019/20

SERVICE DEPLOYMENT

This lab consists of three sessions. Its main goals can be summarized as follows:

```
Contribute to a better understanding of the challenges behind the deployment
of multi-component services, by using some concrete tools and examples.
```

To successfully complete this lab, you will need theory Unit 4 (**Deployment**). You will also need to apply your learnings from Lab2, **ØMQ**, especially the ROUTER-ROUTER *client-broker-worker* (**cbw**) pattern.

This lab identifies 7 milestones, materials for which can be found within their corresponding folders of accompanying material.

The **first session** addresses basic Docker skills, including generation of our base image (0_PREVIO), activities 1 and 2 about the router-router cbw pattern from lab 2, concluding with an introduction to the implementation of `cbw_ftc`, a cbw supporting job classes and a very basic fault tolerance approach.

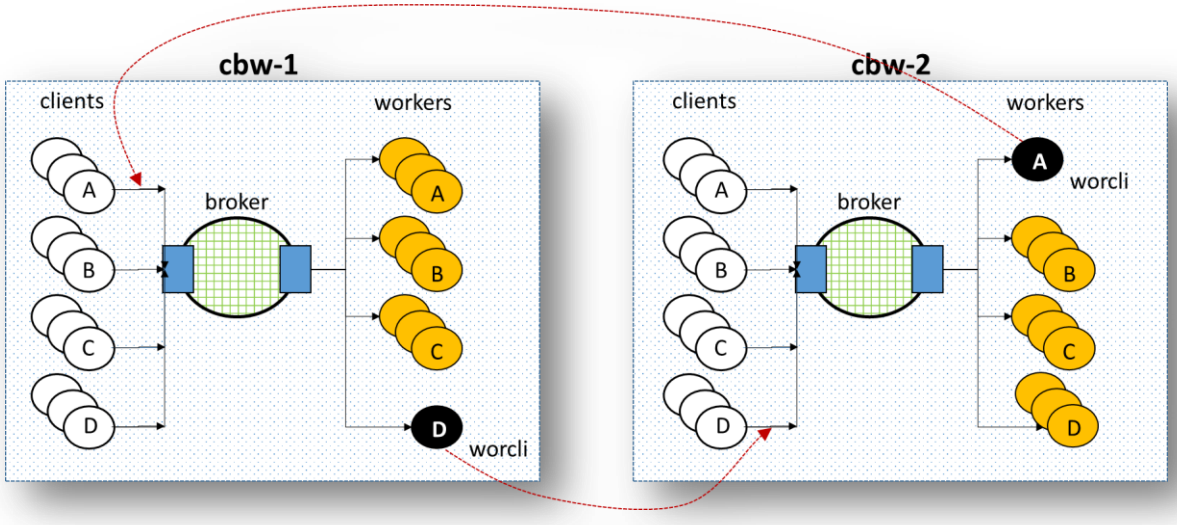
- From this moment on, the code for workers and clients do not need to undergo further changes.

During our **second session** you should complete deployment of `cbw_ftc`, test it, and add a new agent to the service: the `logger` (`4_CBW_FTCL`).

- From this moment on, you will no longer need to modify the broker nor logger either.

You will also need to launch clients running on your desktops (`5_CBW_FTCL_CLEXT`)

During the last session you will need to interconnect two different services



(6_CBW_FTCL_WORCLIEXT) by using a new agent (`worcli`) which proxies between them, without needing modification of the existing component interfaces. To complete this session, you will need to have concluded the previous ones.

What you will need:

1. *tsr_lab3_material.zip* (from "Lessons" in PoliformaT). Also found in the appendices.
   - The virtual machine assigned to you. It already has all Docker tools. If by any chance the docker daemon is not running, you will need to run the following command:

```
systemctl start docker
```

CONTENTS

0 Introduction 4
1 Session 1. First steps with Docker 6
1.1 Building the base image with CentOS, NodeJS and ZMQ 6
1.2 Individual deployment of images for client/broker/worker 6
1.3 Deployment of cbw from Lab 2 7
1.4 New code for high availability and job classes 7
2 Session 2. Intermediate level deployments 8
2.1 Loggin diagnostic information 8
2.2 The logger and its impact on the broker 9
2.3 New component dependencies 9
2.4 Accessing the persistent store from the logger 10
2.5 Combined deployment of the new CBW_FTCL service 10
3 Considerations to take into account before session 3 11
3.1 Redefining the service with decoupled clients 11
4 Session 3. Chaining several cbw systems: *worcli* 12
4.1 Questions about worcli 13
5 APPENDICES 14
5.1 Building image tsr1718/centos-zmq 14
5.2 Appendix 1_A mano 14
5.3 Appendix 2_CBW (basic) 15
5.4 Appendix 3_CBW_FTC (job classes and fault tolerance) 16
5.5 Appendix 4_CBW_TFCL (job classes, fault tolerance and logger) 20
5.6 Appendix 6_CBW_FTCL_WORCLI-EXT 22

0 INTRODUCTION

A service is the result of executing one or more instances of the software components used to implement it. Some of the problems associated to the definition and deployment of services are:

1. **Packaging**, to allow to easily instantiate a component multiple times.
2. C**onfiguration,** to properly set up each one of the deployed component instances.
3. **Component interconnection**. A deployed service forms a distributed system, where multiple agents (the component instances) interact. Interaction happens among *endpoints* defined by each component, communication along paths that must be established on deployment.
4. Lastly, real world scenarios involve a larger variety of interacting agents than those presented in this lab. We will introduce some variations on the exercises that will give us a hint of this complexity.

Many of the concepts exposed in this Lab are based on the scenarios described within the student guide for theme 4 (section 6.5.2).

During this lab we explore some simple ways and technologies to build, configure, connect and deploy components as part of scalable distributed services.

1. Our starting point is Docker as seen in Unit 4. Docker provides a set of tools to prepare and package the software stack needed to properly run an instance of a component.
2. To properly build a component and establish how it should be configured, we will use Docker images, and, more concretely, the *build* method provided by Docker via a ***Dockerfile***.
    - Note that the component code (like any other software) must be configurable, as it needs to adapt to different circumstances on different deployments.
3. To expose the difficulties of interconnecting components we start from the router-router **cbw** from lab 2.
    - Initially we will deploy all components manually, using configuration information as docker command parameters.
    - Then we will automate this activity using the facilities provided by **docker-compose**, which will require us to understand its workings and to learn to specify **docker-compose.yml** files.
    - Finally, we will add **new components** and scenarios, modifying whatever code is needed from the rest of the components, defining new deployment plans.

docker-compose offers some basic scaling functionality. To take advantage of it, components must be properly specified to undergo scaling operations.

4. We will make our scenario a bit more realistic by adding a very basic support for fault tolerance and job classes to our base cbw implementation. From this starting point we will carry out the following activities:
    - Service deployment: the changes carried out introduce different classes of clients and workers.

- Testing the service: verify the behavior when workers fail.
- Decouple the deployment of the logger. Separate the logger component (it does not belong to the same deployment), duplicate cbw_ft_class and deploy all 3 systems: we start with the logger, annotate its IP, and include this information within the deployment of the other 2 services.

The exercises we propose use software (coded in NodeJS) which can be modified to fit our goals.

1 SESSION 1. FIRST STEPS WITH DOCKER

1.1 Building the base image with CentOS, NodeJS and ØMQ

Example 1 in section 6.5.2 from the student guide in Unit 4 summarizes how to generate `tsr1718/centos-zmq`. Here, we show the Dockerfile and command to use in that generation.

1.1.1 Dockerfile

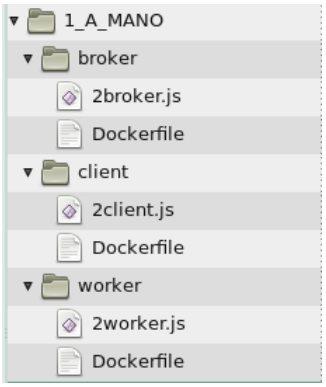
```
FROM centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_10.x | bash -
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN npm install zeromq@4
```

1.1.2 Command

```
docker build -t tsr1718/centos-zmq .
```

Verify that the image exists after the command is executed.

1.2 Individual deployment of images for client/broker/worker



You need to generate the images for the three components, but their code must be adapted from that seen in Lab2:

- The client's code must emit 10 requests[1] before it exits.
- Clients accept 1 command line parameter: the broker socket URL for clients
- Workers accept 1 command line parameter: the broker socket URL for workers
- The broker accepts 2 command line parameters: the socket port for clients, and the socket port for workers. There is also a small change in the code.

These changes are needed to allow component interconnection. The files for the new code are shown prefixed with a "2" in the image above, showing each component within its own folder with its own Dockerfile.

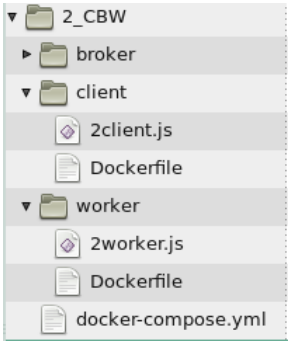
See the corresponding appendix for details.

Next you should generate 3 images (`imclient`, `imbroker`, `imworker`), open 5 terminal windows and run the following commands:

- Terminal 1: **`docker run imbroker`**.
  - Take note of the `imbroker` container's IP, and modify the other component's Dockerfiles.
  - You need to use `docker inspect` on the `imbroker`'s container id.
- Terminals 2 and 3: `docker run imworker`
- Terminals 4 and 5: `docker run imclient`

[1] You must implement the modification

1.3 Deployment of cbw from Lab 2



Section **6.7.4** in the student guide of Unit 4 shows how to prepare a coordinated deployment of several components to build a distributed (**cbw**) service.

We can reuse our previous code modifications, needing only to produce a `docker-compose.yml` file, and create the environment variables ($BROKER_URL) to be used within the new Dockerfile's that must be created for clients and workers.

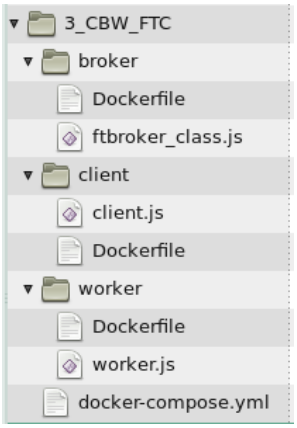
Appendix 2_CBW presents more details. Can you think of problems arising from the usage of the "identity" property?

In order to start 4 clients and 2 workers we should use:

```
docker-compose up --scale cli=4 --scale wor=2
```

1.4 New code for high availability and job classes

In Section 7 of document RefZMQ, we propose several changes in the components of the **cbw** system. Here we choose the union of the following mods:



- Fault-tolerant broker pattern (`ftbroker_class.js`): broker detects and recovers from worker failures (described and solved in section 7.1 of RefZMQ).
- Changes to allow job classes requested by clients, admitted by workers, and supported by the broker. This was mentioned (although it was not solved) in Lab 2.

We will, however, change our starting code for the broker, so that we can more easily introduce further changes. In sum…

"**Fault tolerance**" is achieved using timeouts, establishing a time window during which the broker expects an answer from a worker. If such an answer is not received within the time window (i.e, the timeout expires), the broker assumes the worker has failed, and proceeds to resend the job to another worker of the same class.

To support job classes we could conceivable proceed with any of these alternatives:

- Use a different broker per job class. This could be an interesting option if we could unite the service points in one...
- Have as many internal structures as job classes are admitted by workers (to separate job queues by job class). This has been the option chosen.

Additionally, the new broker expects messages to contain all the information needed (arguments in the client call), being the broker the only one logging information messages to the console.

Let us now have a closer look to the `docker-compose.yml` file. It is evident that we cannot enumerate each combination of client/worker with classes, as that approach would lead us to

an unmanageable amount to cases to consider (e.g., for 2 types of client requests and three kinds of worker jobs, we would need 2x3 cases). Instead, we will have a unique Dockerfile for all clients as well as a unique Dockerfile for all workers.

- Clients and workers will receive their job class through their second command line parameter. To configure the deployment, we will declare as many client components as classes we wish to deploy, and will do the same with workers.
- With this arrangement, we can deploy any combination of instances of various classes of clients and workers using `docker-compose up –scale nnn clientClass –scale mmm workerClass ...`

You can have a look at the code in the annexes. Prepare a deployment and launch 3 clients and 2 workers for each class configured in docker-compose.yml.

To verify that the system behaves correctly you should verify the following,

- Each client receives responses to its requests, and only its requests.
- When a request for a class is pending, no worker for that class should be available.
- Verify failure handling: kill a worker while working on a request, and verify what happens.

Design and implement whatever changes are needed to carry out the above verifications.

1.4.1 Question

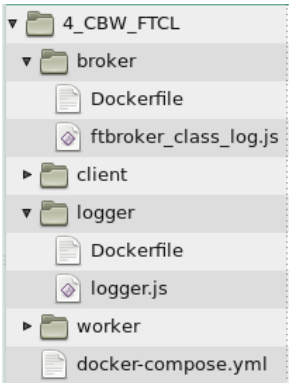
As you can see from the supplied code, the broker discovers the job classes dynamically, as it gets information from clients and workers. This is a great property, as it makes the broker quite flexible.

However, this advantage in the approach to coding the broker gets severely limited by the need to define the job types in the docker-compose.yml file itself.

Try to find a way to avoid explicitly declaring the job types in the docker-compose.yml file, while still being able to determine the number of instances of both clients and workers for each job type.

2 SESSION 2. INTERMEDIATE LEVEL DEPLOYMENTS

2.1 Logging diagnostic information



Components should log messages showing their progress as well as relevant events. The expedite way to actually log such info is to write it to the console. However, it is a better practice to append them in chronological order to some store, for later inspection.

To avoid the complexities of sharing a file among multiple component instances (thus confronting the issue of distributed write access to that file) we choose to design and develop a component capable of receiving those annotations from all component instances: the ***logger***.

- We will use such a service from the broker component, but its use can be easily extended to the rest of components.

Things to take into account:

- The file used by the logger to store the various annotations sent by the components must be PERSISTENT: it must keep its contents between invocations. You will need to use a Docker volume to map that file to a host file, overcoming the ephemeral nature of containers.
- Any component needing to use this logger must have it configured as a dependency to be resolved at deployment time.
- The ØMQ socket type: PULL for the *logger*, and PUSH for the components will suffice.

2.2 The logger and its impact on the broker

The logger and its client components use a push-pull communication pattern (clients push, logger pulls). Its code is available in CBW_FTCL's appendix, and it is very easy to follow for those with some ØMQ knowledge.

In our simple scenario, only the broker uses the logger. Thus, only the broker's deployment configuration needs to take the logger into account. We will need the following elements in the broker

- A PUSH socket to connect with the logger

```
sl    = zmq.socket('push')
```

- The logger's endpoint URL as an argument

```
var lURL  = args[2]
```

- Modify function annotate to send a message to the logger instead of calling console.log.

```
function annotate(prefix, id, cid) {
  var str=util.format(prefix+' %s (class "%s")', id, cid)
  //console.log(str)
  sl.send(str)
}
```

The Dockerfile or the logger is very similar to the others we have already seen. Notice the argument with the path of the containers folder (not the path in the host!!)

You can find the complete code in the appendices.

2.3 New component dependencies

The broker needs to know how to reach the logger. This is similar to the case where clients and workers needed to know how to reach the broker, and can be solved with the same approach: use an environment variable with the URL of the logger. The last line of the broker's Dockerfile will thus be as follows,

```
CMD node mybroker 9998 9999 $LOGGER_URL
```

2.4 Accessing the persistent store from the logger

We need to map the file containing the annotations within the logger's container (`/tmp/cbwlog`) to the host's file system, for which we configure a `volumes`[2] section in the deployment description (docker-compose.yml file).

First, we should create directory `/tmp/logger.log` in the host.

If we were to deploy only this component, independently of the rest of the distributed application, we could employ the docker command directly as this

```
docker run –v /tmp/logger.log:/tmp/cbwlog parameters
```

2.5 Combined deployment of the new CBW_FTCL service

The following fragment of docker-compose.yml shows only the modifications needed to add the *logger*.

```
version: '2'
services:

...

  bro:
    image: broker
    build: ./broker/
    links:
      - log
    expose:
      - "9998"
      - "9999"
    environment:
      - LOGGER_URL=tcp://log:9995
  log:
    image: logger
    build: ./logger/
    expose:
      - "9995"
    volumes:
# /tmp/logger.log DIRECTORY must exist on host and be writeable
      - /tmp/logger.log:/tmp/cbwlog
    environment:
      - LOGGER_DIR=/tmp/cbwlog
```

Note that we can access `/tmp/logger.log` from the host.

Try this: starting from an empty log file deploy a service with 4 type B clients, 2 type B workers, 1 broker and 1 logger.

Question: Describe (without actually running them) what would happen in the following scenarios:

- 2 B clients, 1 B worker, 2 brokers, 1 logger
- 2 B clients, 1 B worker, 1 broker, 2 loggers

[2] Additional information is available in the reference material for theme 4

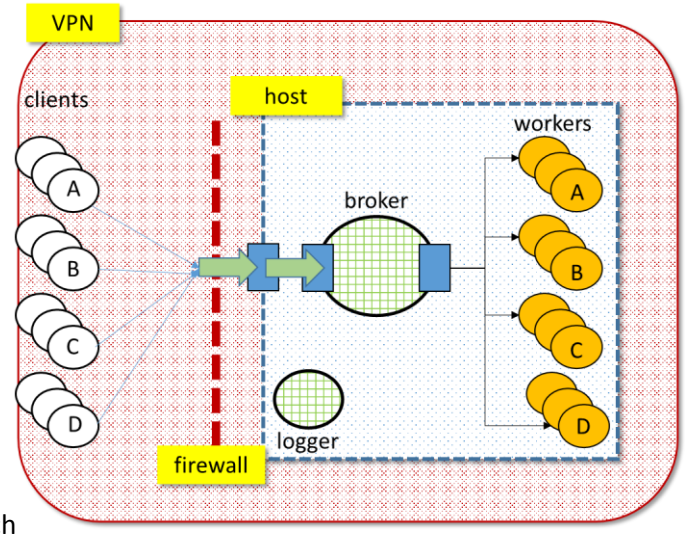
3 CONSIDERATIONS TO TAKE INTO ACCOUNT BEFORE SESSION 3

Our "enriched" **cbw** model is still far from being realistic. In this section we will take steps to get closer to a realistic scenario.

Our first step will reduce the complexity (somewhat) of considering job classes. A second step will reconsider the client's role in our deployment. Finally, a third step will introduce a new component (**worcli**) taking advantage of the work carried out in the previous steps.

3.1 Redefining the service with decoupled clients

The clients are part of the distributed system, but they really do not form part of the distributed service itself. Clients only need to know the API endpoint of the service, but the service needs to know nothing about the configuration of the clients.



What is the endpoint of the service, thus? This is easy: the broker's client URL.

- This URL should be stable in time.
- When clients are on a different host from that which runs the service, we need to somehow route client's requests to the broker. Given that Docker deploys its containers on their own isolated network (by default) we will need to do something to route the requests.

Both problems can be addressed by reserving a port of the host to serve the service. Docker can map this port to the IP:port of the broker when it is deployed.

- If the host's firewall is properly configured, the ports section of `docker-compose.yml` will provoke the needed routing (by port forwarding).
- Note that the VMs are already configured to allow external access to ports 8000 to 8100. We can widen the range to port 9999 running these commands:

```
firewall-cmd --zone=public --add-port=8000-9999/tcp --permanent
firewall-cmd --reload
```

External clients can now connect to the service at URL `tcp://hostIP:9998`, which will forward requests to the broker.

- You can run the clients form any DSIC lab linux desktop, as they have NodeJS+ØMQ installed.
- Note that clients running outside of the host cannot connect to the `logger`.
- If your computer is outside the VPN it should, additionally, access the VPN first.

When checking everything works as expected, consider the following:

1. In the broker section of docker-compose.yml you have added:

```
ports:
"9998:9998"
```

2. Clients are run on the desktops (not the VMs)
3. From the desktop, you should find the IP address of your VM (e.g. with this command: **ping tsr-yourLogin-1920.dsic.cloud**; let us assume that the address is 192.168.105.111)
   - We note again that the IP address of the broker is not routable from outside the host. It is only thanks to the ports section in docker-compose.yml that we can send requests to the host, and have them forwarded to the broker's container.

The client's code (**client_external.js**) is identical[3] to that of the CBW_FTC system. As we run it on the desktop, we need to supply its command line arguments explicitly.

   - Be careful with variable `myID`. Its value would be replaced by the instruction commented out in the original file.
4. Starting from the example's data, and for requests of type B, we should run the client as follows:

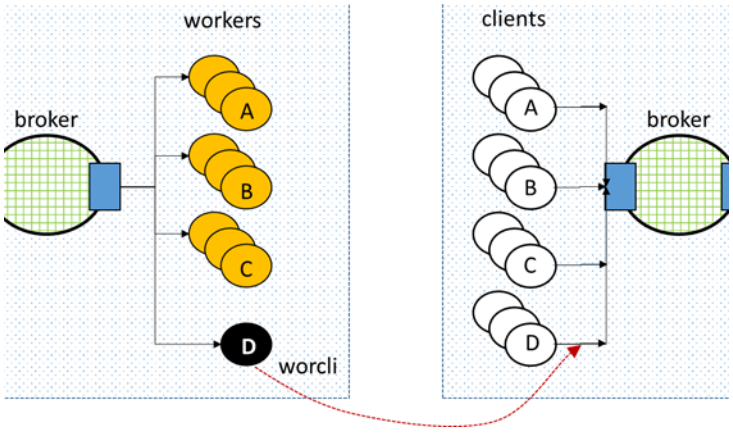
```
node client_external tcp://192.168.105.111:9998 B
```

5. Within the host, the service is deployed using `docker-compose up`.

4 SESSION 3. CHAINING SEVERAL CBW SYSTEMS: *WORCLI*

worcli is a new component, the result of combining the code of worker and client for the same job class, T.

This component should forward requests from a broker bk1, with clients of class T, to broker bk2, with workers admitting job class T.



   - Worcli connects to bk1 as a worker, sending the request as a client to bk2. To return the response, worcli needs to remember the original client identifier.
   - Worcli employs sockets req for each one of the two brokers.

Running worcli requires passing four parameters: bk1's URL, bk2's URL, the **network latency it adds** (argument *delay*) and the job class.

You can find its code in the appendices.

[3] That's why this section has no dedicated appendix

- As with the external clients, we should take care of the myID variable to avoid collisions.

Worcli must be run as if it were an external client: from a desktop.

You should deploy both cbw systems, and then run worcli on your desktop to actually verify that it works properly.

4.1 Questions about worcli

In this section you will find open questions, inviting you to explore various possible approaches to answer them. In all of them we have two deployments, each with 1 broker and multiple workers. Moreover, each deployment runs on a different host.

4.1.1 Question 1

Could *worcli* be deployed as part of a service (via `docker-compose.yml`)?. As an alternative, could you deploy it manually? Provide supporting arguments for your answers. If your answer to the second question is affirmative, explain how you would do it.

4.1.2 Question 2

Assume there is a way to link both service deployments by means of a *worcli*. Assume further that this worcli handles job class D, accepted by no other worker within its source broker.

- Keep in mind that we are using a fault tolerant broker...

Explain why there are significant differences among the following 3 commands. What will the consequences be if the source broker (`192.168.105.111`) receives a client request with job class D.

```
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 200 D
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 1200 D
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 2200 D
```

5 APPENDICES

5.1 Building image tsr1718/centos-zmq

This image has already been built in your VM, but you should check that it is there. The following sections describe how it was built.

5.1.1 Dockerfile

```
FROM centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_10.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 zeromq@4
```

5.1.2 Command

```
docker build -t tsr1718/centos-zmq .
```

5.2 Appendix 1_A mano

5.2.1 2client code

```
01: const zmq = require('zeromq')
02: let req = zmq.socket('req');
03:
04: var args = process.argv.slice(2)
05: if (args.length < 1) {
06:   console.log ("node myclient brokerURL")
07:   process.exit(-1)
08: }
09: var bkURL   = args[0]
10: req.connect(bkURL)
11: req.on('message', (msg)=> {
12:     console.log('resp: '+msg)
13:     process.exit(0);
14: })
15: req.send('Hola')
```

5.2.2 Incomplete client Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./2client.js /myclient.js

CMD node myclient NEED_BROKER_URL
```

5.2.3 2broker code

```
01: const zmq = require('zeromq')
02: let sc = zmq.socket('router') // frontend
03: let sw = zmq.socket('router') // backend
04:
05: var args = process.argv.slice(2)
06: if (args.length < 2) {
07:   console.log ("node mybroker clientsPort workersPort")
08:   process.exit(-1)
09: }
10:
11: var cport = args[0]
12: var wport = args[1]
13: let cli=[], req=[], workers=[]
14: sc.bind('tcp://*:'+cport)
15: sw.bind('tcp://*:'+wport)
```

```
16: sc.on('message',(c,sep,m)=> {
17:     if (workers.length==0) {
18:         cli.push(c); req.push(m)
19:     } else {
20:         sw.send([workers.shift(),'',c,'',m])
21:     }
22: })
23: sw.on('message',(w,sep,c,sep2,r)=> {
24:     if (c!='') sc.send([c,'',r])
25:     if (cli.length>0) {
26:         sw.send([w,'',
27:             cli.shift(),'',req.shift()])
28:     } else {
29:         workers.push(w)
30:     }
31: })
```

5.2.4 Dockerfile broker: build with docker build

```
FROM tsr1718/centos-zmq
COPY ./2broker.js /mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

5.2.5 2worker code

```
01: const zmq = require('zeromq')
02: let req = zmq.socket('req')
03: req.identity = 'Worker1'
04:
05: var args = process.argv.slice(2)
06: if (args.length < 1) {
07:   console.log ("node myclient brokerURL")
08:   process.exit(-1)
09: }
10: var bkURL   = args[0]
11: req.connect(bkURL)
12: req.on('message', (c,sep,msg)=> {
13:     setTimeout(()=> {
14:         req.send([c,'','resp'])
15:     }, 1000)
16: })
17: req.send(['','',''])
```

5.2.6 Incomplete worker Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./2worker.js /myworker.js

CMD node myworker NEED_BROKER_URL
```

5.3 Appendix 2_CBW (basic)

We keep the code for the three components unchanged. Dockerfiles for client and worker are parameterized, resolving dependencies with docker build.

5.3.1 docker-compose.yml

```
version: '2'
services:
  cli:
    image: client
    build: ./client/
    links:
      - bro
    environment:
```

```
      - BROKER_URL=tcp://bro:9998
  wor:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
```

When the Dockerfiles are ready, deploy using `docker-compose up`

5.3.2 Client Dockerfile

Only the last line changes

```
CMD node myclient $BROKER_URL
```

Build using `docker build`

5.3.3 Worker Dockerfile

Only the last line changes

```
CMD node myclient $BROKER_URL
```

Build with `docker build`. Deploy and test

5.4 Appendix 3_CBW_FTC (job classes and fault tolerance)

5.4.1 Client code

```
01: // client in NodeJS, classID must be provided as a parameter
02: // 10 messages and exit!
03: // CMD node myclient $BROKER_URL $CLASSID
04:
05: const zmq = require('zeromq')
06: let req = zmq.socket('req')
07:
08: var args = process.argv.slice(2)
09: if (args.length < 2) {
10:   console.log ("node myclient brokerURL class")
11:   process.exit(-1)
12: }
13: var bkURL   = args[0]
14: var cid     = args[1]
15: var nMsgs = 10
16: var myMsg   = 'Hello'
17: var myID    = "C_"+require('os').hostname()
18: //myID = "C_"+Date.now()%100000 // running without own IP
19:
20: req.identity = myID
21: req.connect(bkURL)
22:
23: req.on('message', (msg) => {
24:   if (--nMsgs == 0) process.exit(0)
25:   else req.send([myMsg,cid])
26: })
27:
28: req.send([myMsg,cid])
```

5.4.2 Client Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./client.js /myclient.js
# We assume that each client is linked to the broker
# container.
CMD node myclient $BROKER_URL $CLASSID
```

5.4.3 Code for ft_broker_class

```
01: // ROUTER-ROUTER request-reply broker in NodeJS.
02: // Work classes.
03: // Worker availability-aware variant.
04: //
05: // As code grows, complexity increases. This version returns to the
06: // original structure with auxiliar functions (sendToWorker & sendRequest)
07:
08: var zmq = require('zeromq')
09: , sc    = zmq.socket('router')
10: , sw    = zmq.socket('router')
11: , util  = require('util')
12:
13: var args = process.argv.slice(2)
14: if (args.length < 2) {
15:   console.log ("node mybroker clientsPort workersPort")
16:   process.exit(-1)
17: }
18:
19: var cport = args[0]
20: var wport = args[1]
21:
22: const ansInterval = 2000
23: var workers = [], clients = [] // clients[] = who{}
24: var busyWks = [] // busy workers
25:          // busyWks[] = tout{}
26:
27: var myID = "B_"+require('os').hostname() // unused
28: //myID = "B_"+Date.now()%100000 // running without own IP
29:
30: function testQueues(cid) {
31:   if (workers[cid]==undefined) {
32:     workers[cid]=[]; clients[cid]=[]
33:   }
34: }
35: function annotate(prefix, id, cid) {
36:   var str=util.format(prefix+' %s (class "%s")', id, cid)
37:   console.log(str)
38: }
39:
40: sc.bind('tcp://*:'+cport)
41: sw.bind('tcp://*:'+wport)
42:
43: annotate('CONNECTED', '', '')
44:
45: // Send a message to a worker.
46: function sendToWorker(msg, cid) {
47:   var myWk = msg[0]
48:   annotate('TO Worker', myWk, cid)
49:   sw.send(msg)
50:   busyWks[myWk] = {}
51:   busyWks[myWk].cid = cid
52:   busyWks[myWk].msg = msg.slice(2)
53:   busyWks[myWk].timeout =
54:     setTimeout(newToutHandler(myWk),ansInterval)
55: }
```

```
56:
57: // Function that sends a message to a worker, or
58: // holds the message if no worker is available now.
59: // Parameter 'args' is an array of message segments.
60: function sendRequest(args, cid) { // (c,sep,m,cid)
61:   if (workers[cid].length > 0) {
62:     var myWk = workers[cid].shift()
63:     var m = [myWk,''].concat(args)
64:    annotate('UNQUEUEING Worker', myWk, cid)
65:     sendToWorker(m, cid)
66:   } else {
67:    annotate('QUEUEING Client', args[0], cid)
68:     clients[cid].push({id: args[0],msg: args.slice(2)})
69:   }
70: }
71:
72: function newToutHandler(wkID) {
73:   annotate('TOUT HANDLER', wkID, '')
74:   return () => {
75:     var msg = busyWks[wkID].msg
76:     var cid = busyWks[wkID].cid
77:     delete busyWks[wkID]
78:     annotate('TOUT EXPIRED', wkID, cid)
79:     sendRequest(msg, cid)
80:   }
81: }
82:
83: sc.on('message', function() { // (c,sep,m,cid)
84:   var args = Array.apply(null, arguments)
85:   var cid  = args.pop()
86:   testQueues(cid)
87:   annotate('FROM Client', args[0], cid)
88:   sendRequest(args, cid) // (c,sep,m,cid)
89: });
90:
91: function processPendingClient(wkID, cid) {
92:   if (clients[cid].length>0) {
93:     var nextClient = clients[cid].shift()
94:     var msg = [wkID,'',nextClient.id,''].concat(nextClient.msg)
95:     sendToWorker(msg, cid)
96:     return true
97:   } else return false
98: }
99:
100:   sw.on('message', function() { // (w,sep,c,sep2,r,cid)
101:     var args = Array.apply(null, arguments);
102:     var cid = args.pop()
103:     if (args.length == 3) {// if (c=='') -> (w,sep,'',cid)
104:      testQueues(cid)
105:      annotate('REGISTERING Worker', args[0], cid)
106:       if (!processPendingClient(args[0],cid)) {
107:         annotate('QUEUEING Worker', args[0], cid)
108:         workers[cid].push(args[0])
109:       //return
110:       }
111:     } else {
112:         var wkID = args[0]
113:         clearTimeout(busyWks[wkID].timeout)
114:         args = args.slice(2)
115:         annotate('TO Client', args[0], cid)
116:         sc.send(args)
117:         if (!processPendingClient(wkID,cid)) {
118:           annotate('QUEUEING Worker', wkID, cid)
119:           workers[cid].push(wkID)
120:       }
121:     }
```

```
122:  })
```

5.4.4 Broker Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./ftbroker_class.js /mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

5.4.5 Code for worker

```
01: // worker server in NodeJS, classID must be provided as a parameter
02: // 1.0 sec service, infinite loop!
03: // CMD node myworker $BROKER_URL $CLASSID
04:
05: const zmq = require('zeromq')
06: let req   = zmq.socket('req')
07:
08: const Tcpu = 1000
09: var replyText = "Done!"
10: var args = process.argv.slice(2)
11: if (args.length < 2) {
12:   console.log ("node myworker brokerURL class")
13:   process.exit(-1)
14: }
15: var bkURL = args[0]
16: var cid   = args[1]
17: var myID  = "W_"+require('os').hostname()
18: //myID = "W_"+Date.now()%100000
19:
20: req.identity = myID
21: req.connect(bkURL)
22:
23: req.on('message', (c, sep, msg) => {
24:   setTimeout(() => {
25:       req.send([c,'',replyText,cid])
26:   }, Tcpu)
27: })
28:
29: req.send(['',cid])
```

5.4.6 Worker Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./worker.js /myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker $BROKER_URL $CLASSID
```

5.4.7 docker-compose.yml

```
version: '2'
services:
  cliA:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9998
      - CLASSID=A
  cliB:
    image: client
    build: ./client/
    links:
      - bro
    environment:
```

```
      - BROKER_URL=tcp://bro:9998
      - CLASSID=B
  cliC:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9998
      - CLASSID=C

  worA:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=A
  worB:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=B
  worC:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=C
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
```

5.5 Appendix 4_CBW_TFCL (job classes, fault tolerance and logger)

5.5.1 Code for logger

```
01: // logger in NodeJS
02: // First argument is port number for incoming messages
03: // Second argument is file path for appending log entries
04:
05: var fs = require('fs');
06: var zmq = require('zeromq')
07: ,   log = zmq.socket('pull')
08: var args = process.argv.slice(2);
09: if (args.length < 2) {
10:   console.log ("node logger loggerPort filename")
11:   process.exit(-1)
12: }
13:
14: var loggerPort = args[0] // '9995'
15: var filename   = args[1] // "/tmp/cbwlog.txt"
16:
17: log.bind('tcp://*:'+loggerPort);
18:
19: log.on('message', function(text) {
```

```
20:   fs.appendFileSync(filename, text+'\n');
21: })
```

5.5.2 Logger Dockerfile

Host requirements: preexisting folder with permissions

```
FROM tsr1718/centos-zmq
COPY ./logger.js /mylogger.js
VOLUME /tmp/cbwlog
EXPOSE 9995
CMD node mylogger 9995 $LOGGER_DIR/logs
```

5.5.3 docker-compose.yml

Client and worker related code remains the same, but we need to consider the logger, and the new dependencies needed so that…

- … the broker can reach the logger (`$LOGGER_URL`)
- … the logger can know the working directory provided by the host (`$LOGGER_DIR`)

We show the relevant fragment of `docker-compose.yml`

```
version: '2'
services:

...

  bro:
    image: broker
    build: ./broker/
    links:
      - log
    expose:
      - "9998"
      - "9999"
    environment:
      - LOGGER_URL=tcp://log:9995
  log:
    image: logger
    build: ./logger/
    expose:
      - "9995"
    volumes:
# /tmp/logger.log DIRECTORY must exist on host and be writeable
      - /tmp/logger.log:/tmp/cbwlog
    environment:
      - LOGGER_DIR=/tmp/cbwlog
```

5.6 Appendix 6_CBW_FTCL_WORCLI-EXT

5.6.1 Code for worcli-ext

```
01: // worcli
02: // invoked with "node worcli bk1URL bk2URL delay class"
03: // all 5 parameters are mandatory
04: var zmq = require('zeromq')
05: , rw = zmq.socket('req')
06: , rc = zmq.socket('req')
07:
08: var args = process.argv.slice(2)
09: if (args.length < 4) {
10:   console.log ("Usage: node worcli bk1URL bk2URL transfer_delay class")
11:   console.log ("Redirects bk1's class requests to bk2 broker, increasing delay ms")
12:   process.exit(-1)
13: }
14:
15: var w2bk = args[0] // worcli connected to bk1 as a worker
16: var c2bk = args[1] // worcli connected to bk2 as a client
17: var myID  = "WC_"+require('os').hostname()
18: //myID = "W_"+Date.now()%100000
19: var delay = parseInt(args[2]) // transfer delay, in ms
20: var cid   = args[3]
21: var pendingClient
22:
23: rw.identity = myID; rw.connect(w2bk)
24: rc.identity = myID; rc.connect(c2bk)
25:
26: rw.on('message', (c,sep,m) => {
27:   pendingClient = c // only one waiting client, so we don't need a queue
28:   setTimeout(()=>{
29:       rc.send([m, cid])}
30:       , delay/2); // 50% forwarding
31: })
32:
33: rw.send(['', cid])
34:
35: rc.on('message', (m)=> {
36:   setTimeout(()=>{
37:     rw.send([pendingClient,'',m,cid])}
38:     , delay/2); // 50% returning
39: })
```