Maeva Siga
Clément Fauvre

PROYECTO DE LABORATORIO

Project about the interaction among three main ZeroMQ microservices.

Introduction

This project is an example of a Faas architecture, that is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.

Here the Faas architecture is composed of three microservices called Frontend, Queue and Worker.

For each part there is a node file and a dockerfile to create the Docker image. In fact the whole project is deployed with Docker.

The general goal of the application is to receive the ID of the clients and to print them.

Frontend

The frontend is the entry point of the app. The client sends his ID and the frontend sends this ID to a queue in order to be elaborated later. In order to do that, the frontend is divided in two parts. In the first part there is the a **REST API** that listens to the port 8000 of the Docker container (5000 of the host machine) and will receive the clients request thanks to a POST middleware (http://localhost:5000/id).

The second part is a **ZMQ req socket** that will send the client requests to one of the three queues randomly.

In the dockerfile about this microservice, 'frontendDockerfile', we expose the relevant port and we install the necessary dependency (zeromq and express).

Queue

The queue is where all the clients ID go and they wait to be elaborated by the workers. There are indeed three instances of a queue, but they all do the same thing. Having more instances of a queue allows us to speed up the process if we have a lot of requests, because more queues can work parallely.

Each queue is a **rep ZMQ socket** that is bound to a different port. They receive the ID, and put them in the list of the ID that still have to be elaborated by the workers. There are 2 workers that are **req ZMQ socket**. If there are workers available the queues send them one of the IDs to elaborate. Then they wait for the worker to end his job before putting it back on the list of the available workers.

If no worker is free the queue will just wait until one of them is free again.
Once the worker is done, it will send a message to the queue so it can know for what ID the work is already done and it will communicate to the frontend. So we can keep track of the client requests that are ready (or not eventually).

In the dockerfile about this microservice, ‘queueDockerfile’, we expose the relevant port and we install the necessary dependency (zeromq).

Worker

The worker file does the “job” on the client's ID and sends it back to the queue. The job in our case is just printing the ID of the client. There are two workers that work in an asynchronous way in order to speed up the process even more. It’s obvious that more workers means more ID printing at the same time and this means a significant growth of the speed. The workers are **rep ZMQ sockets** that received the IDs from the queue and print them on the console also saying which worker did it (the one or the two).
When the print is done, they send a message to the queue to notify them that they finished the work on the ID the queue sent them. In this way we keep track of the IDs and we can be available again for the next ID.

In the worker file there is a function that overrides the function console.log(). We added this in order to have a file that contains all the log (“debug.log”), so that we can make sure that there is no failure, and that the work has been done for every request.

In the dockerfile about this microservice, ‘workerDockerfile’, we expose the relevant port and we install the necessary dependency (zeromq).

Test

The two tests are simple node files that use the library **request** to execute some http post request to the REST API of the front-end microservice. The first one just sends 11 requests in a row with different IDs, and the second one 100 (to test the scalability of the app).
The test showed that putting more queues and more than one worker improves the quality of the service, because at first we tried it just with one queue and one worker and it was indeed slower.
To test the app we just have to execute the following command in the terminal:

```
> sudo docker-compose build && sudo docker-compose up
```

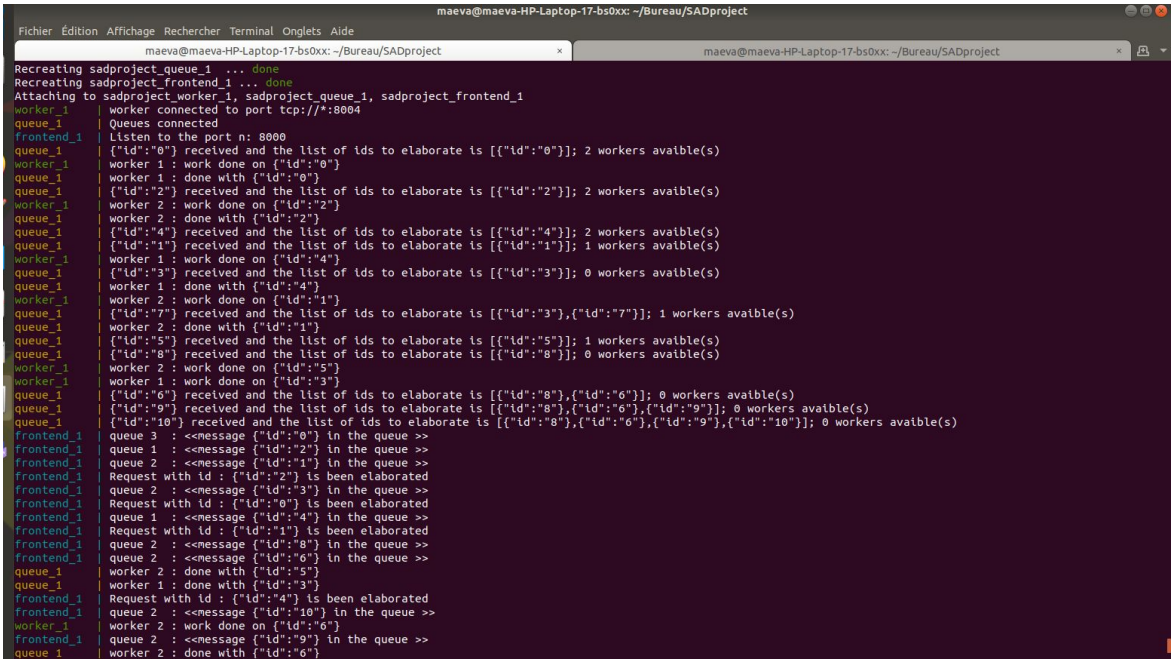
Then in another terminal:

```
> node test1.js
```

or

```
> node test2.js
```

Below, an example of execution:



Conclusion

In conclusion this application shows how Faas services work. The three microservices communicate correctly between them and the tests don't show any kind of worker failure (or any other failure of any type). The communication is fluid and the app keeps track of the worker used and of the client request. The workers' job is just logging in the console but it could be easy to write a function that does something with the client requests, and that retrieves the result of the job. Even if the worker job is very basic, the application allows a good scalability: if the requests appear to be a lot more numerous, we just have to add more and more workers.

One extra that could have been interesting to explore is to allow the queues to create more workers or remove some just based on the number of pending client requests.