Project B2: FaaS management

Danilo D'Amico
Università degli Studi di Roma Tor Vergata
Rome, Italy
danilo.damico@students.uniroma2.eu

***Abstract*—This report details a project undertaken for the course *Sistemi Distribuiti e Cloud Computing*. The project focuses on deploying a single given function, coded in a specific language and passed through a URL, as a Function as a Service (FaaS). The system manages this function within a local container and employs a decision policy to determine when to offload it to the cloud.**

I. Introduction

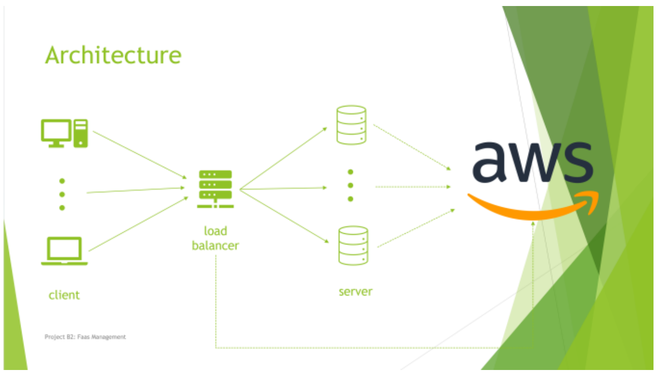
Functions as a Service (FaaS) are a cloud computing service that offers users the ability to deploy and invoke a function without needing to allocate specific resources for it. This project aims to offer a user the ability to use a function by simply deploying it to a Redis database and passing the URL to the system. The system itself, when unable to deal with the number of requests on its own, delegates its requests to another serverless platform to ensure scalability.

II. System Architecture

A. Languages and Frameworks

The system was developed in Python and local deployment was managed through multiple Docker containers. Requests offloading is made from inside the docker containers to an AWS Lambda exposed through an API Gateway. AWS CloudWatch was used to control the correct execution of the remote function. Communication both between the client and containers and the containers themselves is handled through XML-RPC. The necessary container images and parameters are defined in a docker-compose.yml.

B. Architecture



This system has been designed and tested on a single machine. Its client and server both operate leveraging the same Docker Engine.

The project consists of three main actors:

- A client responsible for preparing a Redis database housing an arbitrary Python function.
- The central system itself.
- Multiple clients capable of initiating concurrent requests to the load balancer.

The system itself is composed of a load balancer and a cluster of local servers. The load balancer is tasked, on startup, with deploying the function to make available on AWS and instantiating the servers declared in its environment variables. After this preparatory phase, it is ready to serve requests. The load balancer achieves elasticity by dynamically scaling server capacity, flexibly adjusting the number of servers up or down in response to the number of concurrent client requests. After reaching the maximum number of servers allowed by its configuration, the load balancer offloads every additional request to the cloud.

Servers receive a request from the load balancer and serve them. A server is a local container that communicates with the load balancer inside a Docker network. They initially try to serve the request locally, however, if the request takes more time than indicated in the configuration file, it offloads it to the cloud instead.

III. Implementation Insights

A. General insights

This chapter will focus on implementation details regarding the system described in the previous chapter. The code that will be described is available on this GitHub repository: https://github.com/DaniloDamico/SDCCproject.

The Python project is articulated into three modules:

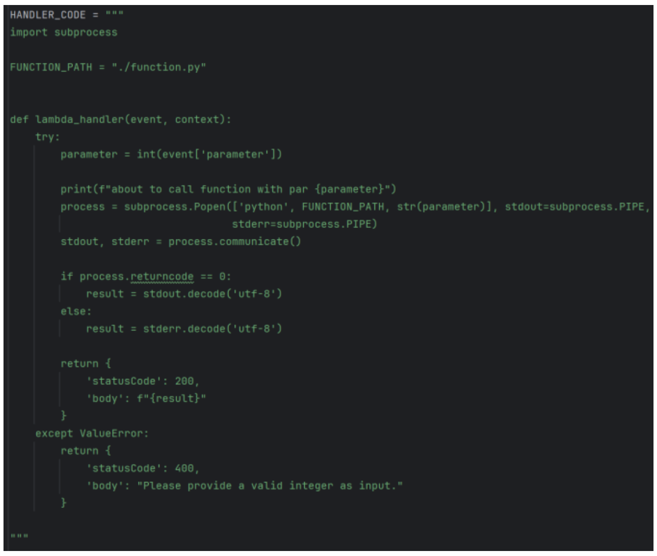
- *client*, containing the code relative to the client that instantiates the Redis container (*add_faas.py*) and the relative *Dockerfile*. Finally, *client.py* contains the code to make a request to the load balancer.
- *load_balancer*, consisting of the load balancer's implementation (*load_balancer.py*) and the *Dockerfile* for the container it resides in.
- *server*, containing the *Dockerfile* and *server.py* to create a container that acts as a local server.

In the main module, there is a *docker_setup.py* that instantiates the system. In the setup phase, *docker_setup.py* writes a *docker-compose.yml* passing the variables specified in an *.env* file. The docker compose is comprised of a docker network, a *loadbalancer* service and a *server* service. The script builds the *server* service into a docker *server-image* and launches the *loadbalancer*. When the *loadbalancer* creates a new server, it simply launches a new *server-image* with the appropriate environment variables.

B. *The load balancer*

The load balancer is the center of the system. It is also, however, a single point of failure and the centralized element through which servers are managed and requests are dispatched. Improvements to the system should look into either decentralizing it or adding redundancy and recovery from failure. It is composed of an initialization phase and a service phase.

In the initialization phase, it has to either create or update an *AWS Lambda* to serve the arbitrary Python function, expose it through an *API Gateway* and create the number of servers indicated in the *.env* file. It is not possible, however, to simply run any Python script inside *AWS Lambda*. Instead, the service requires a precise structure: a *lambda_handler* method that starts the execution and a structured response. To make any Python script available on AWS without modifying it, the load balancer uploads two functions to the cloud: the function itself, in a file called *function.py* and an *handler.py* can be called by *Lambda*. The handler runs the function in a subprocess and adds its return value to its own return value.



```
HANDLER_CODE = """
import subprocess

FUNCTION_PATH = "./function.py"


def lambda_handler(event, context):
    try:
        parameter = int(event['parameter'])

        print(f"about to call function with par {parameter}")
        process = subprocess.Popen(['python', FUNCTION_PATH, str(parameter)], stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE)
        stdout, stderr = process.communicate()

        if process.returncode == 0:
            result = stdout.decode('utf-8')
        else:
            result = stderr.decode('utf-8')

        return {
            'statusCode': 200,
            'body': f"{result}"
        }
    except ValueError:
        return {
            'statusCode': 400,
            'body': "Please provide a valid integer as input."
        }

"""
```

Once the initialization phase has been completed, the load balancer switches to a service phase. In here, it creates a threaded XML-RPC server and creates another thread to manage its elasticity. The server forwards requests according to a round *robin policy*. In case of failure, it forwards the request to the next server the scheduling will indicate and creates a new thread that tries to kill and recreate the non-responding server from scratch, making the system resistant to server failures. The elasticity thread, on the other hand, every *elasticity_sleep* seconds wakes up and checks the number of active connections. If there are more than the number of available servers, it increases them up to 2 * *number_of_servers* (the initial parameter set by the user). If, conversely, the number of active connections is lower than then number of servers, it decreases them down to *number_of_servers*.

IV. RESULTS

The local offloading policy chosen for local servers is based on the assumption that serverless computing means that AWS would adjust its allocated resources automatically in order to best serve the incoming requests. To test this hypothesis, the system was tested using the $O(n^2)$ Fibonacci algorithm. What resulted, instead, was the cloud having significantly less computing power than the local container, leading to worse performances than simply running all computations locally. It must be noted that, due to the empirical observations discussed in class relative to the CPU resources assigned by AWS depending on memory allocated, the Lambda was allocated with the maximum amount of RAM possible (3008MB).

For this reason, computationally intensive functions should be run locally whenever possible and the limit on containers raised or even removed altogether. These considerations are heavily machine specific and depend on the hardware configuration of the laptop used to develop and test the system. Different configurations may lead to different results and different optimal policies.

The system is best suited to running many low resources tasks, which is the ideal use case for serverless computing.

CONCLUSIONS

The proposed system manages to offer a serverless Python function. It manages to react to multiple requests and can be scaled significantly by fine tuning its configuration.

A. *Critical aspects*

- As indicated in section III. B. the load balancer is a single point of failure. Decentralizing it or adding redundancy would make the system as a whole more robust.
- The load balancer needs access to a valid credentials file to interact with AWS. This could be mitigated by complete control over AWS IAM policies and roles but is inevitable in the AWS Academy Leaner Lab environment.

B. *Future Improvements*

- The API Gateway is not protected by an API Key. The URL of the Gateway is generated at runtime and never revealed outside the docker network; however an API Key could be added to avoid unexpected AWS costs.
- The *round robin* scheduling policy could be replaced by the *least number of connections* policy.
- In case of a non-working server, the load balancer hangs up for some seconds before reporting the error. This could be mitigated by having a

heartbeat thread that checks the state of servers on a regular basis

- The current elasticity model could be replaced by a more advanced prediction engine that better anticipates future loads, keeping containers warm when needed.