Project B2: FaaS Management

Sistemi Distribuiti e Cloud Computing
Università degli Studi di Roma Tor Vergata
Danilo D'Amico
0320389

Index

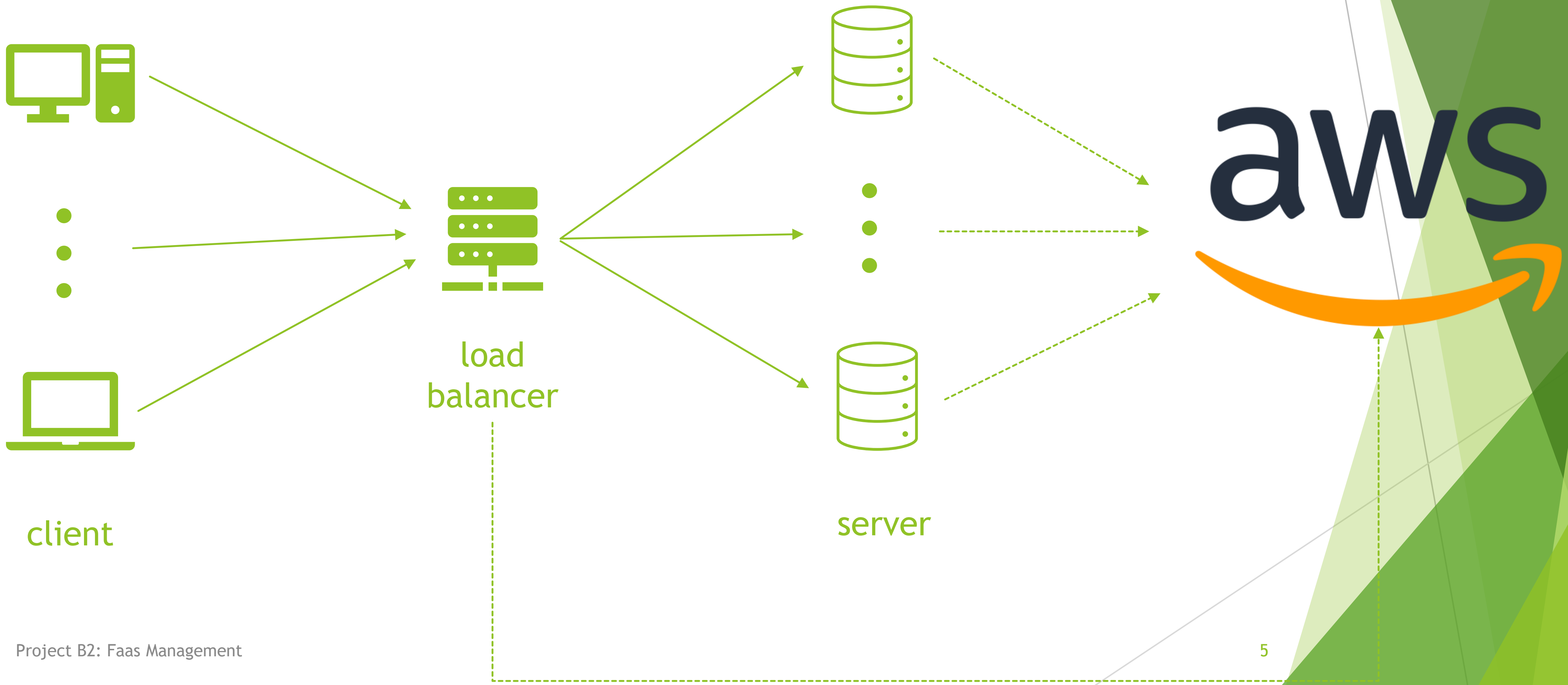
Aim

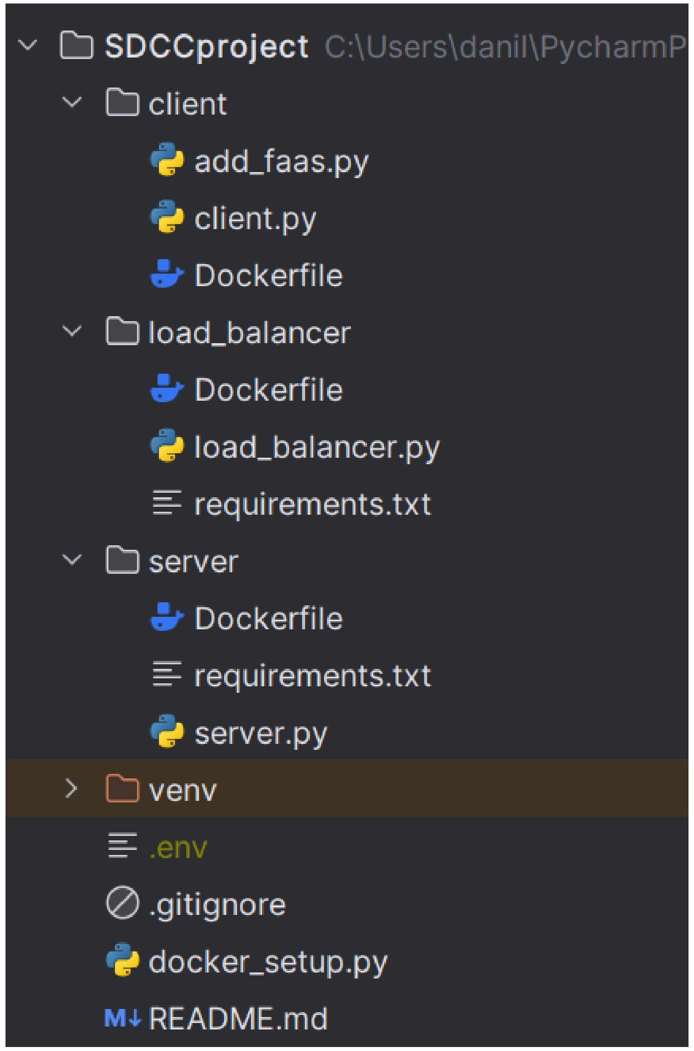
- Manage single functions written in a given programming language inside a container
- Offload to Cloud according to a local decision policy
- Deployment using Docker and AWS Lambda for function offloading

Tools

- The languages and framework utilized for the system developement are:
  - Python for the system logic
  - Docker Compose for local deployment of contaienrs
  - AWS Lambda for remote serverless computing
  - AWS API Gateway for RESTful access to AWS Lambda
  - AWS CloudWatch for remote execution monitoring
  - XML-RPC for communication between clients and containers locally

Architecture





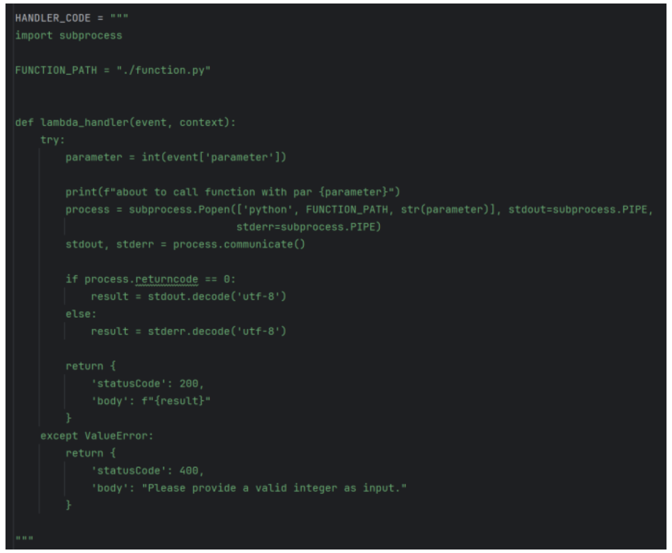
```
SDCCproject C:\Users\danil\PycharmP
  client
    add_faas.py
    client.py
    Dockerfile
  load_balancer
    Dockerfile
    load_balancer.py
    requirements.txt
  server
    Dockerfile
    requirements.txt
    server.py
  venv
  .env
  .gitignore
  docker_setup.py
  README.md
```

Implementation – General Insights

- **Client**: the code relative to the client that instantiates the Redis container and the client that makes requests to the load balancer.
- **Load Balancer**: the load balancer's implementation
- **Server**: the source code for one of the many local servers managed by the load balancer
- **Docker_setup.py**: the script to dynamically create the docker-compose to start the system. Builds the server-image, creates the docker network and starts the load balancer

Implementation – Load Balancer

- The center of the system
- also, a single point of failure and the centralized element through which servers are managed and requests are dispatched.
- Initialization Phase:
  - Starts AWS Lambda & API Gateway
    - Cannot pass an arbitrary function to Lambda directly! => *handler.py*
  - Starts NUMBER_OF_SERVERS Servers
- Service Phase:
  - Threaded XML-RPC server
    - Round robin
    - Server recovery from failure
  - Elasticity Thread
    - (NUMBER_OF_SERVERS , 2* NUMBER_OF_SERVERS)



```
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."
        }

"""
```

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.
- 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
- These considerations are heavily machine specific
- The system is best suited to running many low resources tasks, which is the ideal use case for serverless computing.

Conclusion – Critical Aspects

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.

Conclusion – 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.