Auto-Scaling for Cloud Microservices

Final Report

ECE 422

Youssef Ismail, Ali Muneer, Rupin Bapuji

Abstract

The purpose of this project is to provide a web server that will scale according to the volume of client requests. The server will monitor the average client request time and horizontally scale (creating/destroying) instances to keep the average time in a specific, predetermined zone. The system defines acceptable response time thresholds, and if response times exceed the upper bound, the auto-scaler scales out (adds an instance). Vice versa, if the response times fall below the lower bound, the auto-scaler will scale in (destroying an instance). The benefits of this project is to ensure an adequately responsive and cost-efficient approach to web applications. The system is tested using a normal distribution of workload (users), which demonstrates its ability to optimize both cost and performance, depending on the situation.

Tools & Technologies

Development Environment

Python: As the primary programming language. It's versatile and well-supported for web and cloud applications. Further, the given code is already implemented in Python, which will improve the cohesiveness of our code. It also has a well-documented library that will allow us to interface with our instance of docker. This will allow us to scale our microservice programmatically.

Flask: A lightweight and easy-to-use Python web framework for our web microservices. It is well-documented and will also make inter-container communication simple.

Docker Swarm: For container orchestration to manage and scale the microservices efficiently. Services will be able to be scaled up and down via container duplication.

Redis: As a datastore for caching and storing the amount of hits on the web server. This will be kept as its own separate service, to ensure this datastore acts as “global” for our web app.

Cybera Infrastructure: Our web application will be hosted on cloud infrastructure, namely Cybera Cloud. There will be 3 Ubuntu servers, with one designated server as the manager node. All 3 will join the docker swarm to distribute our microservices across.

Version Control and Collaboration

Github: In order to collaborate in an Agile environment, we shall be using Github for collaboration. Its interactive GUI facilitates the use of code review and pair programming. Further, its branch and merge functionality allows us to engage in iterative development. Separate branches can be made for each user story to simulate continuous integration.

Testing Tools

Python Client Simulation: We shall write a script written in Python that will simulate a realistic load on our web service. It will scale the amount of concurrent requests on a gaussian basis, while monitoring the response times.

Project Management Tools

Github Projects: For tracking our progress we shall use the embedded project view within Github. This will tightly integrate our codebase with our project management system to allow us to track our overall progress on a commit-by-commit basis.

Communication Tools

Discord: To communicate with our team and distribute our tasks accordingly we shall use Discord as an informal communication medium. Sprint planning and interim progress will be posted and discussed there.

Github Issues: More formal documentation of bugs and features will be done on different issues, by “commenting” on issues to keep a trackable line of communication.

Design

Client

We implemented **scaling_client.py**, a file that simulates a number of concurrent users. This implementation is largely based on **http_client.py**, a file given in the project specification. Each user is represented by a thread, which sleeps for a specified amount as part of its workload. We also added retry functionality to the send request code that was provided to us. Currently, the client will try a maximum of 5 times before backing off.

Autoscaler

Scaling

The **monitor** function holds the bulk of the functionality. It is responsible for analyzing recent response times and scaling if necessary. The function runs in an infinite loop, checking if the recent response time is within an acceptable range. If not, the service either scales up or down. It then sleeps for a predetermined time (0.5 seconds). The **updateTimes** function is responsible for keeping track of the response times of the server. When the server completes a task for a user, it posts the time to a route (/time), from which the function retrieves and appends to a global array.

Graphing

The **getTimes** function is used to post a JSON object to the route /plots. This JSON object contains data for three tables: average response time, workload, and number of replicas. The **graph** function is used to populate the /graph endpoint on the web server. A response object is generated with predefined HTML/JavaScript. This object calls the /plots endpoint to retrieve the data for the graphs. It then uses Plotly, a graphing library, to plot the three graphs. These graphs update in real time (updates happen every 0.5 seconds, according to our predefined interval).

Web App

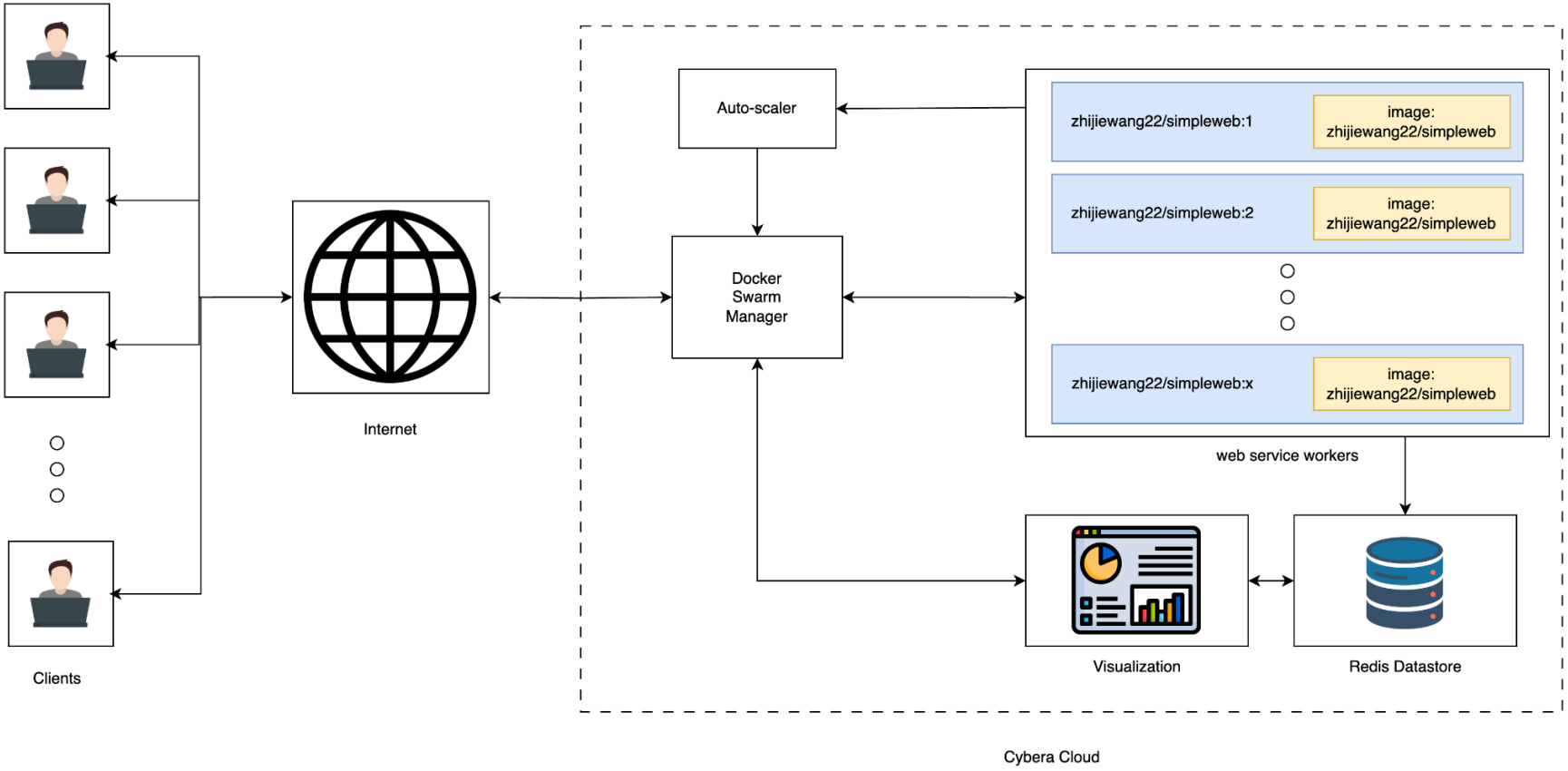
The functionality of the web-app is largely unchanged from the project specification. We implemented the post functionality of the computation. Once the Redis **hits** value is incremented, we post the computation time to the /time endpoint, allowing the server to retrieve and store the data.

User Stories

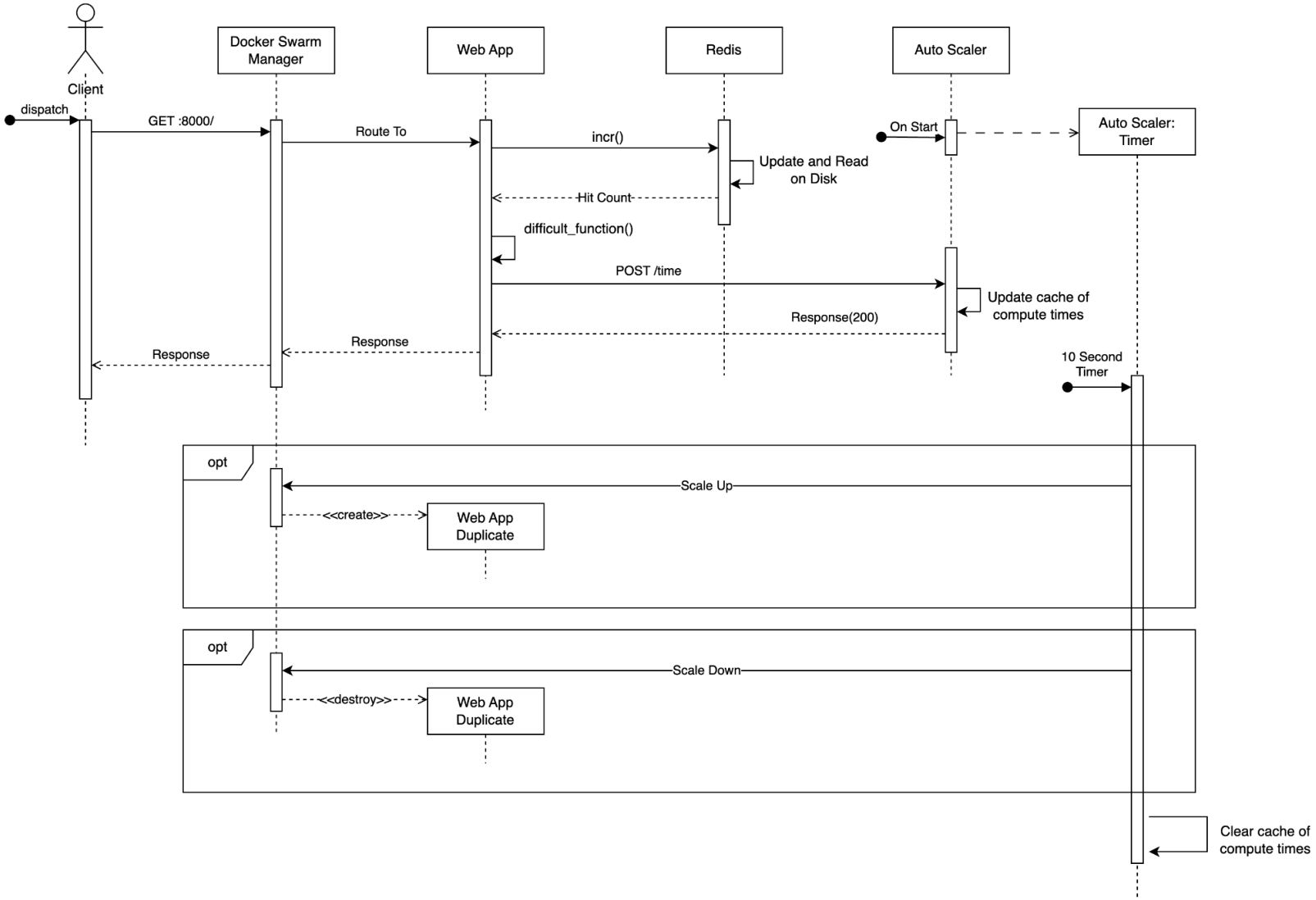
1. As a business owner, I want the autoscaler to scale efficiently, so that it can minimize costs and unnecessary resource expenditure.

2. As an administrator, I want to be able to tune scaling parameters, so that the autoscaler can allocate resources based on predefined conditions.

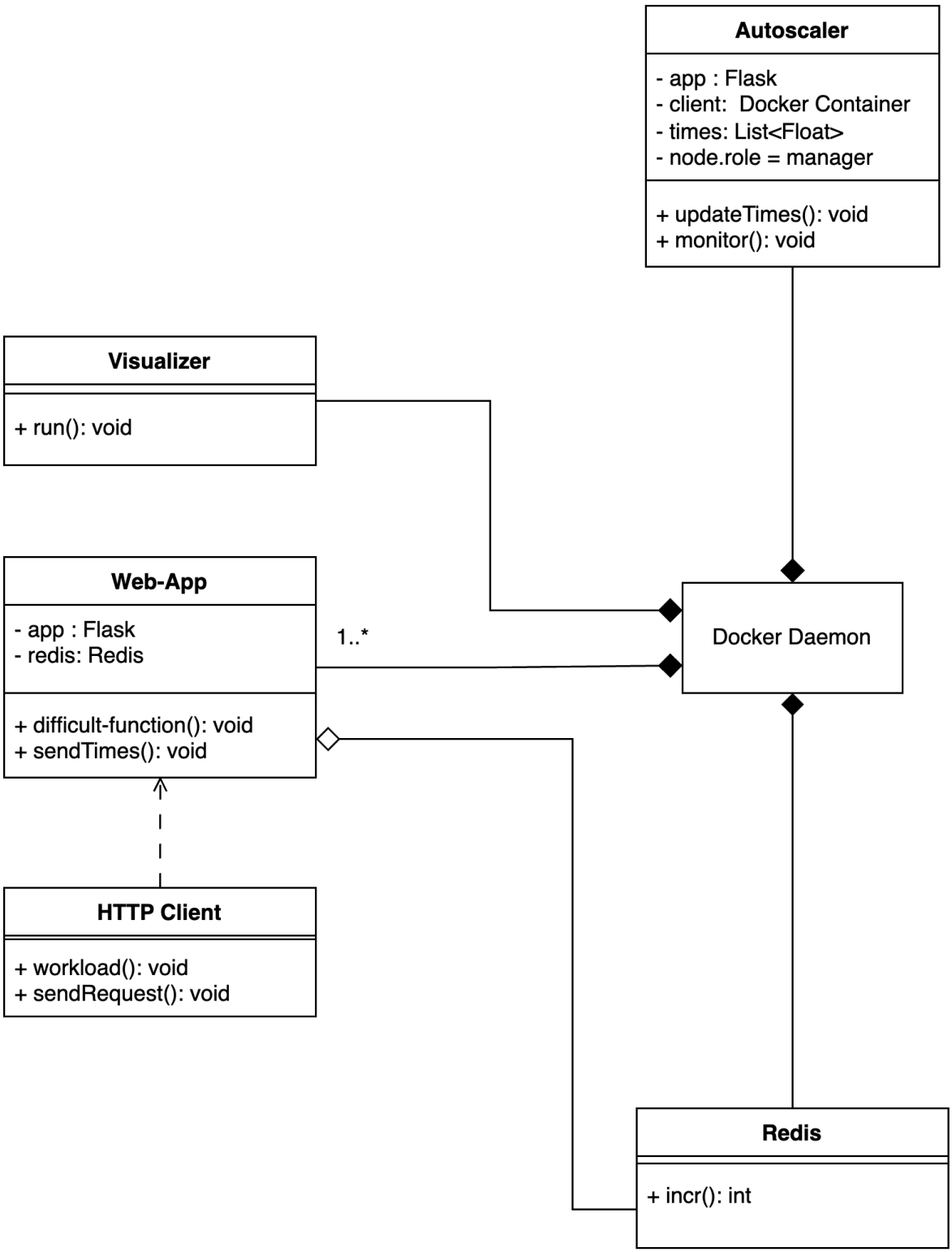
Architecture Diagram



Sequence Diagram



Class Diagram



Deployment

1. Create 3 VMs on Cybera cloud with the following specifications:
    a. Use Ubuntu 18.04 or Ubuntu 20.04 as the image for all VMs.
    b. You need one of these VMs to run the client program for which you may use m1.small flavor. Let's call this VM as the Client_VM.
    c. For the other two VMs, please still consider m1.small flavor. These two VMs will construct your Swarm cluster.
    d. You need to open the following TCP ports in the default security group in Cybera:
    22 (ssh), 2376 and 2377 (Swarm), 5000 (Visualization), 8000 (webapp), 6379 (Redis)
2. On the Client_VM run
    $ sudo apt -y install python-pip
    $ pip install requests

3. Then, you need to install Docker on VMs that constitute your Swarm Cluster. Run the following on each node.
    $ sudo apt update
    $ sudo apt -y install docker.io

4. Now that Docker is installed on the two VMs, you will create the Swarm cluster.
    a. For the VM that you want to be your Swarm Manger run:
    $ sudo docker swarm init
    b. The above init command will produce something like the bellow command that you need to run on all worker nodes.
    $ docker swarm join \
    --token xxxxxxxxxxxxxxxxxx \
    swarm_manager_ip:2377
5. Run the following to deploy your application:
    $ sudo docker stack deploy --compose-file docker-compose.yml alpha

6. You should now see four services, a visualization microservices, a web application, redis, and the autoscaler.

7. Now login to your Client_VM and run the following:
    $ python3.5 http_client.py swarm_manager_ip 1 1

Demo the Autoscaler

1. Login to the client vm and run:
   $ python3 scaling_client.py 10.2.7.210 4 1 # <ip n_users think time>
2. This will run a scaling client that creates up to 1, 2, 3, 4.. N users every 60 seconds and then scales down backwards.
3. Then connect to the vpn and go to **swarm_manager_ip:8001/graph**. This will display stats about the request in a live graph view.

Conclusion

In conclusion, the auto scaling engine has been successfully implemented. The microservice application is fully able to scale in and out, depending on the workload present. This service has been tested under a normally distributed number of users, and has performed as expected. By setting hard upper and lower bounds for response time, the server ensures that a client task will always be completed in an acceptable zone. This project serves to demonstrate the needed balance between efficiency and cost-effectiveness, which is demonstrated by autoscaling.

References

[1] Z. Wang, “ECE-422-Proj2-StartKit,” GitHub, 2023. [Online]. Available: https://github.com/zhijiewang22/ECE422-Proj2-StartKit