Autoscaling for Cloud Microservices

ECE 422

Tobi Ogunleye (1636868)
Ren Wen Lim (1593841)
Chengxuan Li(1631060)

Abstract

This paper outlines a reactive auto-scaling engine based in Python for a cloud microservice application. The application is meant to be deployed onto the Cybera Rapid Access Cloud infrastructure using Docker microservices. It also includes a webpage for the real-time plot of some of its response and scalability metrics.

Introduction

Auto-scaling is a crucial component of modern cloud-based applications, allowing them to adjust to varying workloads and maintain optimal performance and reliability. In this report, we will explore the development and implementation of an auto-scaler that uses horizontal scalability to adjust the application based on user workload. The auto-scaler will monitor response times and adjust the number of microservice instances to maintain an acceptable range of response times, defined by upper and lower thresholds. If response times exceed the upper threshold, the auto-scaler will scale out the application to ensure reliability and performance. Conversely, if response times fall below the lower threshold, the auto-scaler will scale in the application to optimize operational costs on the cloud. By implementing this self-adaptive application, we aim to achieve optimal performance and cost-effectiveness simultaneously.

Design - Methodology and Technologies

For this project we decided to use Python, due to the ease of use with Flask, Docker and Redis - which are all essential frameworks for our project. The visualization was done using Javascript and CSS for the frontend, which are fairly commonplace for web-development. We also made use of the locust API which helped to provide varying rates of requests so that we could adequately test out the application response with non-linear loads.

Load Management

We chose a load management approach that uses a queuing algorithm to determine the number of replicas required to handle the current traffic. The algorithm considers the arrival rate of visitors and the average response time to calculate the traffic intensity, and then calculates the number of replicas needed using the following formula:

$$s \geqslant \max(1, \rho + \sqrt{\rho})$$

Where
s = number of parallel servers
$\rho = \frac{\lambda}{\mu}$ = traffic intensity = ratio of mean arrival rate and mean service rate.

The LoadBalancer class is responsible for implementing the load management algorithm. It uses a Redis database to store the number of visitors and the average response time, which are later used to calculate the traffic intensity. The class also uses the Docker API to manage the number of replicas. The run method of the LoadBalancer class periodically calculates the number of replicas required based on the current traffic intensity and the number of replicas currently running. If the number of replicas required is greater than the number of replicas currently running, the LoadBalancer class scales up the replicas. If the number of replicas required is less than the number of replicas currently running, the LoadBalancer class scales down the replicas. This ensures that the application can handle the current traffic while minimizing the cost of running unnecessary replicas.

The load management approach implemented by the code above has several benefits. Firstly, it is efficient in scaling up and down replicas based on the current traffic intensity. This ensures that the application can handle the current traffic while minimizing the cost of running unnecessary replicas. Secondly, the use of a queuing algorithm ensures that the application can handle the load in a systematic way. Finally, the use of a Redis database to store the number of visitors and the average response time ensures that the load management algorithm is based on accurate and up-to-date data.

Visualization

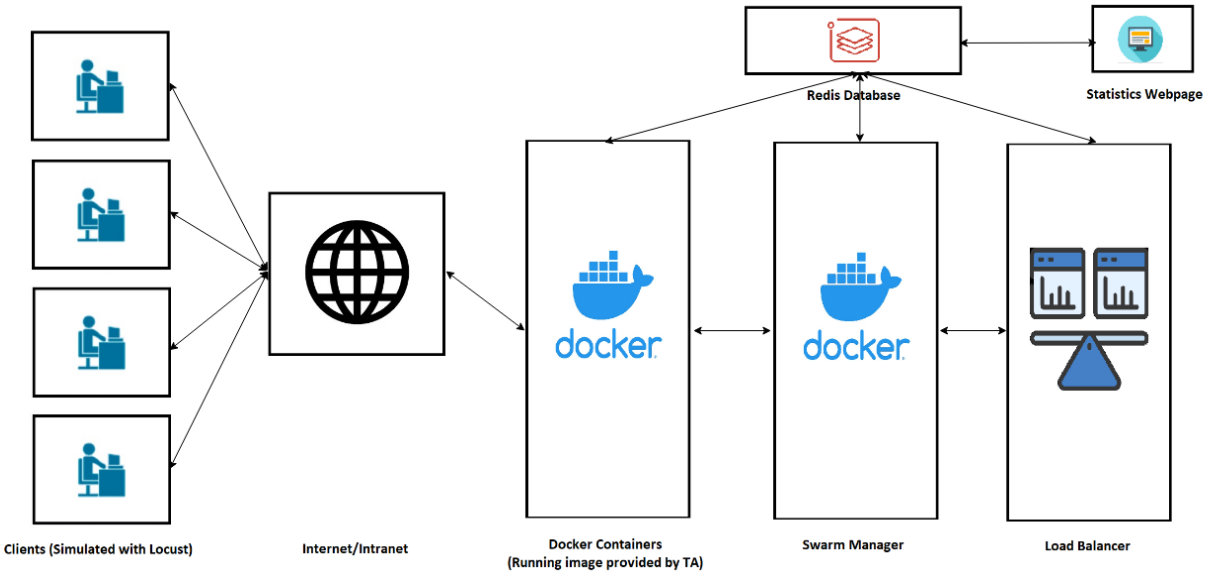
This code is an example of front-end implementation using the Chart.js library to visualize real-time data on a web page. The code creates three charts, each with a different dataset and options, and attaches them to HTML elements using their IDs.

The first chart, 'replicas', visualizes the number of replicas of a service over time. The chart type is 'line', and the dataset is initially empty. The x-axis is set to be of type 'realtime', with a duration of 20000 milliseconds, a refresh rate of 1000 milliseconds, and a delay of 2000 milliseconds. The onRefresh function is defined to fetch data from a server at 'http://10.2.6.145:3000/replicas' and push it to the dataset, along with the current timestamp.

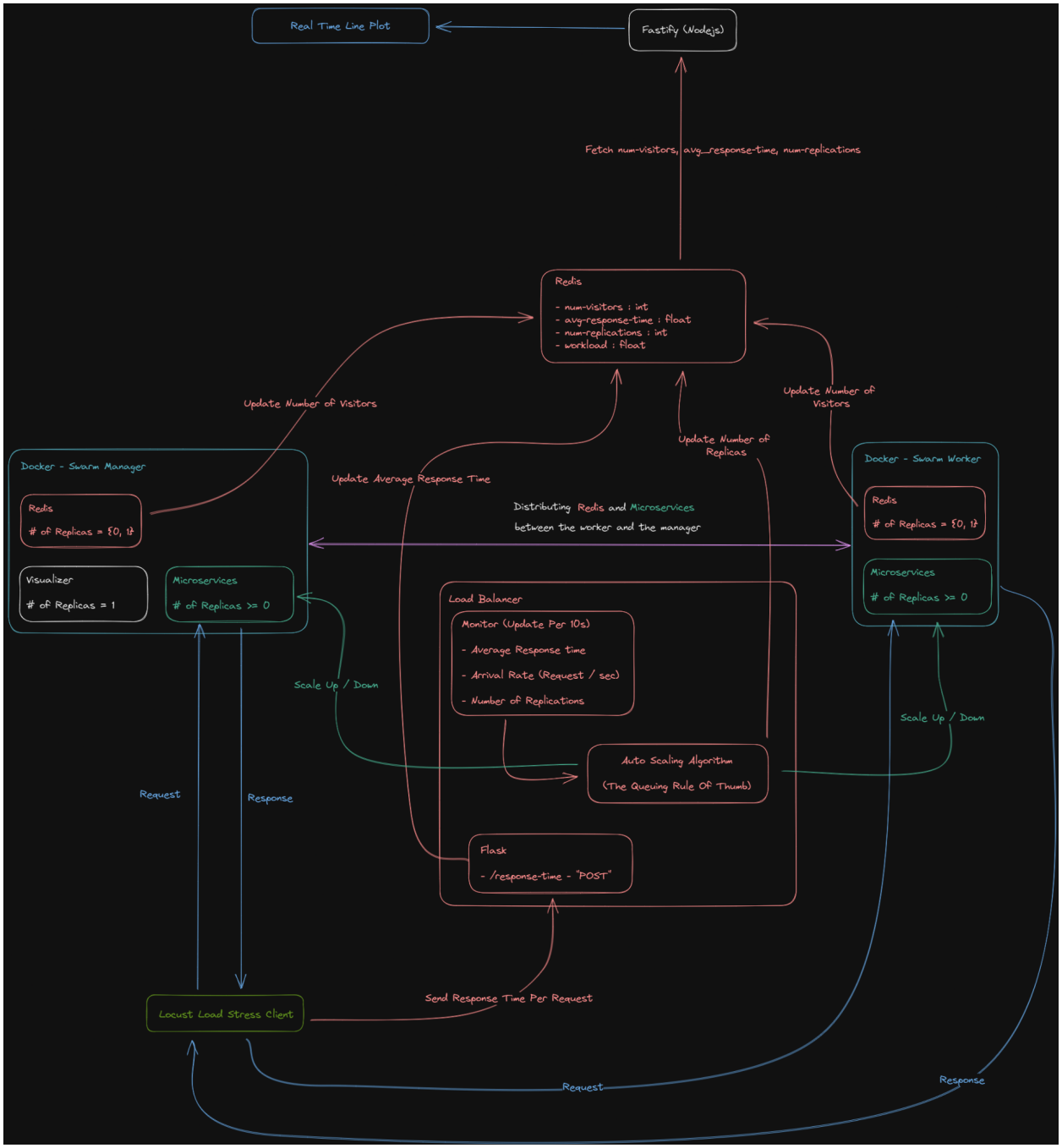
The second chart, 'time', visualizes the average response time of a service over time. It has similar options to the first chart, except that the data is fetched from 'http://10.2.6.145:3000/avg_response_time'.

The third chart, 'workload', visualizes the workload of a service over time, represented as the arrival rate of requests within a 10-second window. It has the same options as the first two charts, with the data fetched from 'http://10.2.6.145:3000/workload'.

Design Artifacts

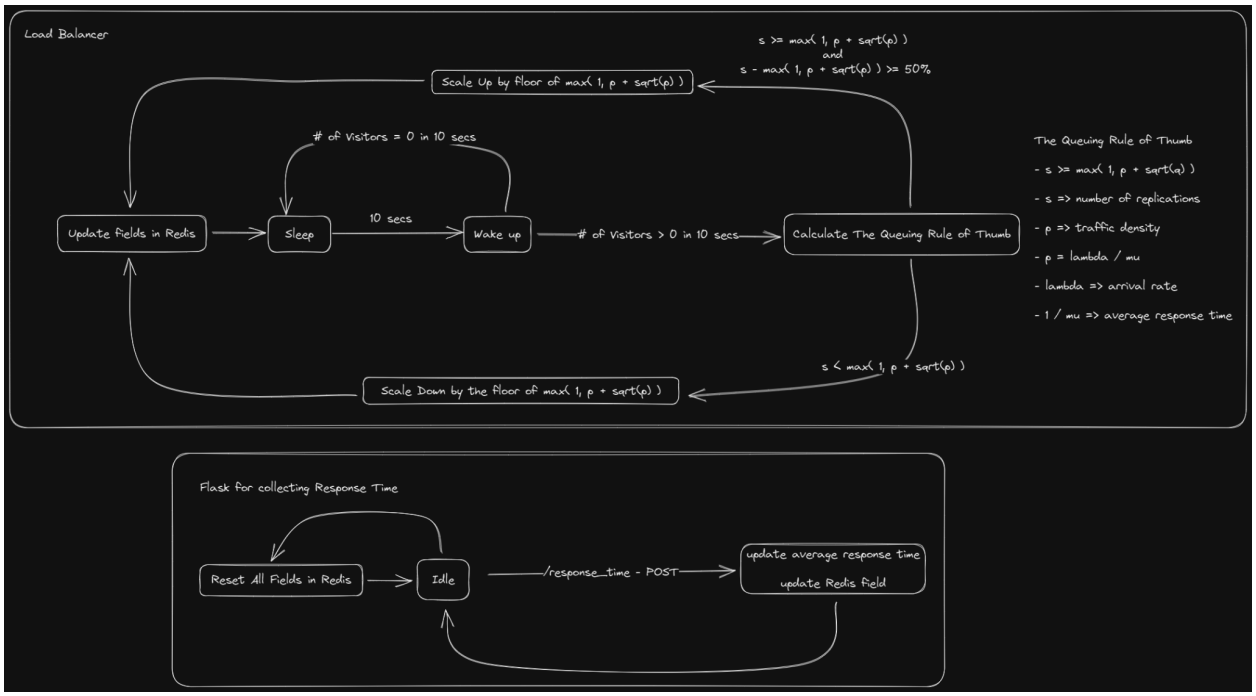


This high-level architecture diagram provides an overview of the interactions between the clients and various components of our project.



This UML diagram showcases the core functions and associations between our applications. The Redis database stores all of the necessary variables for executing our auto-scaling logic and displaying our real-time performance statistics. The LoadBalancer class handles the auto-scaling logic and updates the Redis database with the new statistics, as well as updating the Swarm Manager with the new number of containers for it to scale to. The Swarm Manager and Workers respond to the varying

workloads provided by the Locust stress test, scaling their number of active containers based on the values provided by the LoadBalancer, and updating the Redis database accordingly.



This state-diagram provides a high-level overview of the LoadBalancer class's logic loop in monitoring the rate of requests and scaling the active number of containers, as well as the logic loop of our Flask application's response time statistics collection.

#Auto-scaling algorithm pseudocode

#Key Parameters:

#Checking interval in seconds
BALANCER_SLEEP_TIME = 10

#Algorithm pseudocode

While True:

    Wait for BALANCER_SLEEP_TIME

    If current_no_of_visitors = 0 and previous_no_of_visitors > 0:

        current_no_of_visitors =0; previous_no_of_visitors = 0

    Get average_response_time from redis database
    traffic_intensity = (current_no_of_visitors - previous_no_of_visitors)/BALANCER_SLEEP_TIME*
    average_response_time

    factor = max(1, traffic_intensity $+ \sqrt{\text{traffic\_intensity}}$)

    If number_of_containers >= factor and (number_of_containers – factor)/number_of_containers >=0.5:
        Set number_of_containers = floor(factor)

    else if number_of_containers < factor:
        Set number_of_containers = ceil(factor)

As mentioned in our load balancing section, our auto-scaling algorithm calculates the number of active containers it should have based on the formula $number\_of\_containers = max(1, traffic\ intensity + \sqrt{traffic\ intensity})$.
It therefore does not require input parameters for scaling thresholds, and the only key parameter here is the waiting time between checks. This has been set to 10s as in our testing, it provides the best scaling response times while avoiding the response ping-pong effect.

Deployment Instructions

Users will have to first go through the setup process outlined here: https://github.com/zhijiewang22/ECE422-Proj2-StartKit
Once the setup is complete, the user will have to clone the repo at: https://github.com/Dekr0/autoscaling-web-app. The clone should be made at the swarm manager and client VMs. The client directory can be removed from the swarm manager VM and vice versa. Once the program has been cloned, you will need to ensure that you install all the necessary libraries such as flask, redis and Chart.js. After this is done,

navigate to the /app directory and redeploy the docker container using the command “make deploy”

User Guide

Users will need 3 instances of the swarm manager VM terminal to execute the project.
To start up the load management algorithm, navigate to the /load_manager directory and type in “make”.
To start the server for the performance graphs, navigate to the /backend folder, and type the command “node server.js”

Finally to start up the front-end of the graph webpage, navigate to the /graph directory and enter the command “make”

With all the swarm manager tasks started, open the client VM, navigate to the /client directory and enter the command “make”. You will then need to go to the locust webpage at: http://10.2.6.215:8089/ and enter the host url: “http://10.2.6.145:8000/”. Locust will then start hitting the host endpoint with the specified levels of requests automatically. Then the graph will begin to show the changes in number of replications, average response time and the workload.

Conclusion

This Python-based auto-scaling engine project was successfully completed, meeting all its requirements. Users are able to see in real time the adjustments made to produce a bell-shaped response curve based on the workload on the cloud microservices. The program appropriately scales the number of active containers up or down based on the workload, while largely maintaining the average response time to within the threshold. The visualizations are also accurate and real-time, allowing users to see an accurate representation of the current load and number of microservices running.

References

1. https://people.revoledu.com/kardi/tutorial/Queuing/Queuing-Rule-Of-Thumb.html
2. https://www.devgraph.com/resource/what-is-a-load-balancer-definition-explanation/#:~:text=Weighted%20Response%20Time%20is%20a,calculate%20the%20application%20server%20weights.
3. https://locust.io/
4. https://www.chartjs.org/