Auto-Scaling for Cloud Microservices

ECE 422 - RELIABLE SECURE SYSTEMS DESIGN

Authors:

Chanpreet Singh

Gurbani Baweja

Abstract

Our objective in this project was to develop a reactive auto-scaling engine for a cloud-based microservice application. The focus of the auto-scaler is on horizontal scaling for the web microservice in a system composed of two microservices. The aim is to maintain user request response times within an acceptable range while optimizing performance and minimizing cost. The auto-scaler adjusts the application by comparing response times with established standards. This results in a balanced system that can adapt to changes in workload. This report outlines the technologies and tools that we used for accomplishing this project, design artifacts with explanation, and deployment instructions along with a user guide.

Introduction

Microservices have played a key role in the evolution of cloud computing, transforming the way applications are developed and deployed by improving scalability. However, managing the scalability of microservices can be difficult due to changing workloads. This project addresses this challenge by developing a reactive auto-scaling engine. The engine adjusts the application based on user request response times, ensuring optimal performance and reducing costs. The engine focuses on the horizontal scaling of the web microservice in a two-microservice system, keeping user request response times within acceptable limits while optimizing performance and cost.

During the development process, we carefully selected appropriate technologies, such as the Python request library, Redis, and the Plotly library, to accurately monitor response times and enable the auto-scaler to adjust the application based on workload. This report provides a detailed explanation of the technologies used and includes a high-level architectural view of the auto-scaler's features, a state diagram showing the states, events, and actions in the autoscaler, and the pseudocode of the auto-scaling algorithm. To make the project user-friendly, we have also included deployment instructions and a user guide in this report.

Technologies

We used the request python library to send HTTP requests to the docker swarm manager as it enabled us to send requests and calculate the average computation time, giving us an estimation if the replicas need to be increased or decreased to reduce or increase the current computation time. Then we used Redis, as the webapp was storing the number of hits in the Redis database, so for our workload graph we needed that information to analyze the workload with hits. We used the Plotly library of Python to make all the graphs which were required for this project. In addition, we also used os and time library to execute commands in swarm manager to change the replicas to handle the traffic and time library supported in computation time calculation.

We even considered other technologies such as using matplotlib instead of Plotly, however, our aim for live graphs motivated us to use Plotly, however considering that our VMs don't have graphics, thus we were ultimately only able to make live graphs but then view them on local machine only when execution ended.

Design

Our AutoScale application is standalone and does not require any modification in the webapp that was provided as a starter kit. We decided to make it standalone so it could be more widely used and doesn't need docker images to be changed. The following diagram shows a high-level architectural view of the auto-scaler's features.

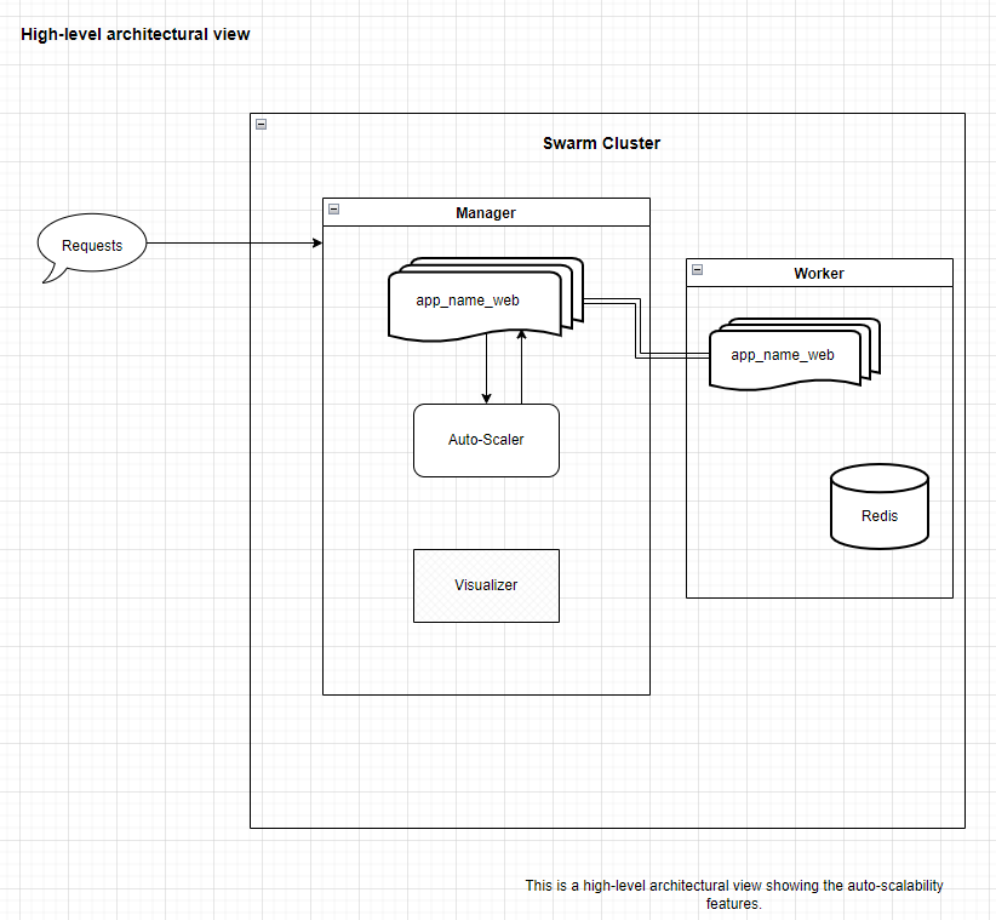

Fig 1: high-level architectural view of the auto-scaler's features

When AutoScale.py is run, we start sending one request every second to compute the compute time for each request to be processed by webapp. As this approach created artificial traffic on swarm, we decided to set a minimum number of webapp replications to 2, so the compute time could be the same for autoscaler on or off when one request is coming per second.

After every 10 requests compute time, we find the average compute time and then evaluate the average compute time to decide if the application size needs to be increased or decreased. The following state diagram clearly depicts the states, events, and actions in the autoscaler.

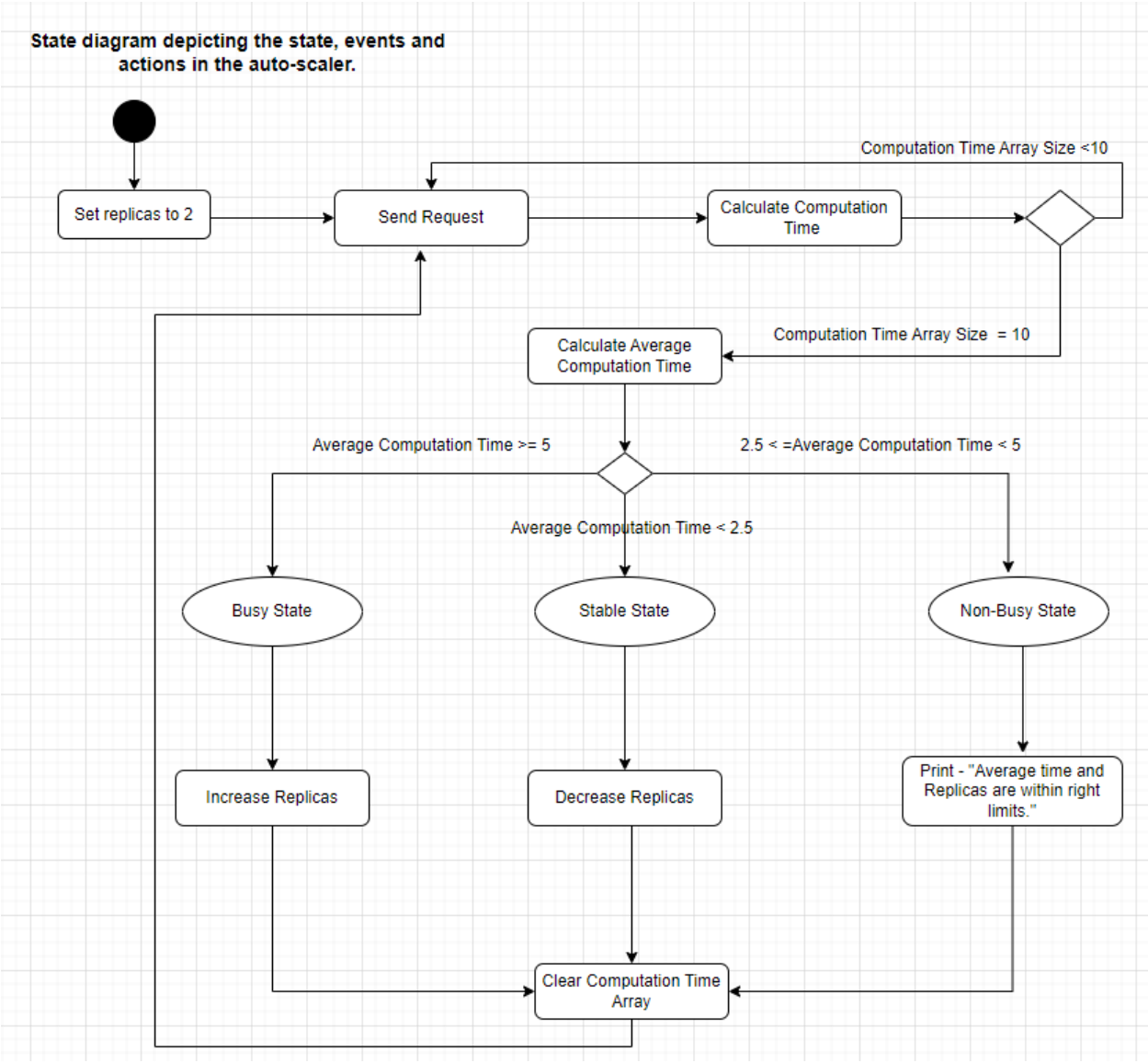

Fig 2: A state diagram showing the states, events, and actions in the autoscaler

Scaling Policy:

- If the average scaling time over 10 requests from autoscaler is above 5 second, then we increase the application size with one additional replica.
- If the average scaling time over 10 requests from autoscaler is under 2.5 second and the minimum number of replicas is more than 2, then we decrease the application size with one additional replica.
- If the average scaling time over 10 requests is between 2.5 to 5, then we dont make any change to the application size i.e. replicas.

After analyzing the average scaling time and then making changes (if needed) as per scaling policy, we restart sending requests and analyze if the change made improved computation time or do we need to make another increment or decrement in application size again.

It is important to note that we do wait for 10 request times, and keep a gap of 2.5 sec as acceptable computation time, for the reason that we monitor the traffic for a reasonable time before we make any change to application size, it is also important to note that os command takes 5 - 8 sec to make the change, so having application resized very frequently shall lead to alot of unnecessary delays. Thus we found the current scheduling policy to be reasonable in handling auto-scaling tasks. We could have opted to calculate the average over less than 10 response times like maybe for 3 or 6, but we found that 10 is more reasonable to fully analyze the impact of the last application size increment or decrement before making the decision to make further changes to the size. Thus our application scale with increment or decrement of 1. If we had changed the webapp image and stored computation time in Redis, that could have been an easier way to implement the same auto-scaling, however then our AutoScale code would have relied on image changes and would have been customized for only this image that store computation time in redis, and won't work on docker image that don't store computation time. In its current form, our implementation is more universal to autoscale different types of webapp images and support the scaling as workload. We took the project as a challenge to make an autoscale that scales the provided webapp application. A pseudocode of the auto-scaling algorithm is attached below to provide an insight for the algorithm that we used.

```
                                        Pseudo Code

Docker service scale to 2 replicas.


New ending while loop:

        Record start time.
        Send request to 10.2.1.38:8000
        Record end time

        Compute time (end time - start time) and store in array

         if(length of array equal or greater than 10):

                Compute the average time over last 10 recorded times.

                if( average time is greater or equal to 5):
                        Docker service increment to +1
                Else if (averate time is lower than 2.5 and replicas greater than 2):
                        Docker service decrement by -1
                Else:
                        Do nothing
        Empty the compute time array

        Wait for 1 second and repeat the while loop.
```

Fig 3: Pseudocode of the auto-scaling algorithm

We have attached one graph in the appendix that shows how increased workload led to increasing computation time and our application scaled, then when the workload was reduced, the computation time had anyway reduced due to increased application size, the autoscaler then reduced the application size.

Deployment Instructions

To set-up the AutoScaler on Cybera:

1. Follow the README instructions on our GitHub to set up swarm manager and cybera VMs: https://github.com/Chanpreet-Singh-UofA/Auto-Scaling-for-Cloud-Microservices-

2. Clone the code from GitHub git clone
   https://github.com/Chanpreet-Singh-UofA/Auto-Scaling-for-Cloud-Microservices-.git

3. Add autoScaler.py to the swarm manager VM by running following command
   scp -i path/to/manager.pem -r path/to/autoScaler.py ubuntu@<ip_of_manager_vm>\:

To run the AutoScaler on Cybera using our VMs:

1. Connect to the Cybera instance using the ssh.
   ssh -i path/to/manager.pem ubuntu@2605:fd00:4:1001:f816:3eff:fe1c:6b08

2. Run the python script to turn AutoScaler ON:
   python3 autoScaler.py

3. To stop AutoScaler press Control+C, and then you can access the plots as 'plot.png' file.
   scp -i path/to/manager.pem ubuntu@10.2.1.38:/home/ubuntu/plots.png
   path/to/where/you/want/to/save/graph/locally

To send requests:

1. Either go to http://10.2.1.38:8000/
   or
2. Use Client VM
   ssh -i path/to/r1.pem ubuntu@2605:fd00:4:1001:f816:3eff:fecf:5a5a
   Python3 http_client.py

User Guide

Run the python script (autoscaler.py) on the Manager VM and the AutoScaler will be turned ON, it dont need any further instructions. To stop the AutoScaler by interrupting the application by pressing control+C.

Conclusion

Our project has successfully designed a responsive auto-scaling mechanism for cloud-based microservice applications. Our auto-scaling engine focuses on horizontally scaling the web microservice within a dual-microservice system, ensuring that user request response times remain within suitable boundaries while improving performance and minimizing costs.

To develop this adaptive auto-scaling engine that can handle fluctuating workloads, we used technologies such as the Python Requests library, Redis, and the Plotly library for data visualization. The auto-scaling engine is implemented on the Cybera architecture through Docker microservices.

In addition to the auto-scaling engine, we have also provided clear diagrams, including a high-level architectural view, state diagram, and pseudocode for our auto scaling algorithm, to aid in understanding our design for this project. Overall, our design provides a reactive auto-scaling engine for a cloud-based microservice application that meets the project's requirements.

References

- Project Description provided on Eclass
- Starter Kit

Appendix