ECE 422

Reliability Auto-Scaling Project

Team Squeaky Hotdogs

Muhammad Fiaz, Zong Lin Yu, Gurbir Sandha

January 23, 2024

Introduction

A key component of cloud computing is the ability for scaling applications automatically depending on workload. For DevOps/SysOps engineers, manually scaling applications is not a feasible solution. In this assignment, we implemented an auto-scaling engine for a cloud microservice application.

Tools and Technologies

Containerization: We used containerization to automate the setup process for the environments of the different microservices. Using containerization allows us to use pre-defined images and configuration, improving the portability of the different services. We specifically used Docker as our containerization service. From a security perspective, it improves the security of our service due to the separate file systems. From a reliability perspective, Docker allows us to more easily scale our service when paired with Docker Swarm, and secondly, quickly reboot the container upon any software failures.

Docker Swarm: For our auto-scaler service, we centered our program around the usage of Docker Swarm. This is an organization tool that coordinates multiple instances of Docker Swarm Nodes. Together, these nodes recreate the function of a singular service. Docker Swarm allows for dynamically scaling up and down a service by increasing and decreasing the number of Nodes in a Swarm. In our case, we scale our web microservice based on the service's average response times. The web microservice is simulated on a cluster of 2 virtual machines, one of them serving as the manager, and the other running a Swarm worker. We will use the Docker Python SDK to interact with our Docker Swarm.

Redis: Response times are stored in a Redis datastore, which is a local key-value storage service. The web microservice will interface with Redis through its Python SDK, where we will keep the response times from the last 10-20 seconds. Our autoscaler will then read and reset this number of hits. Our auto-scaler code is written in Python, using the Redis SDK mentioned above.

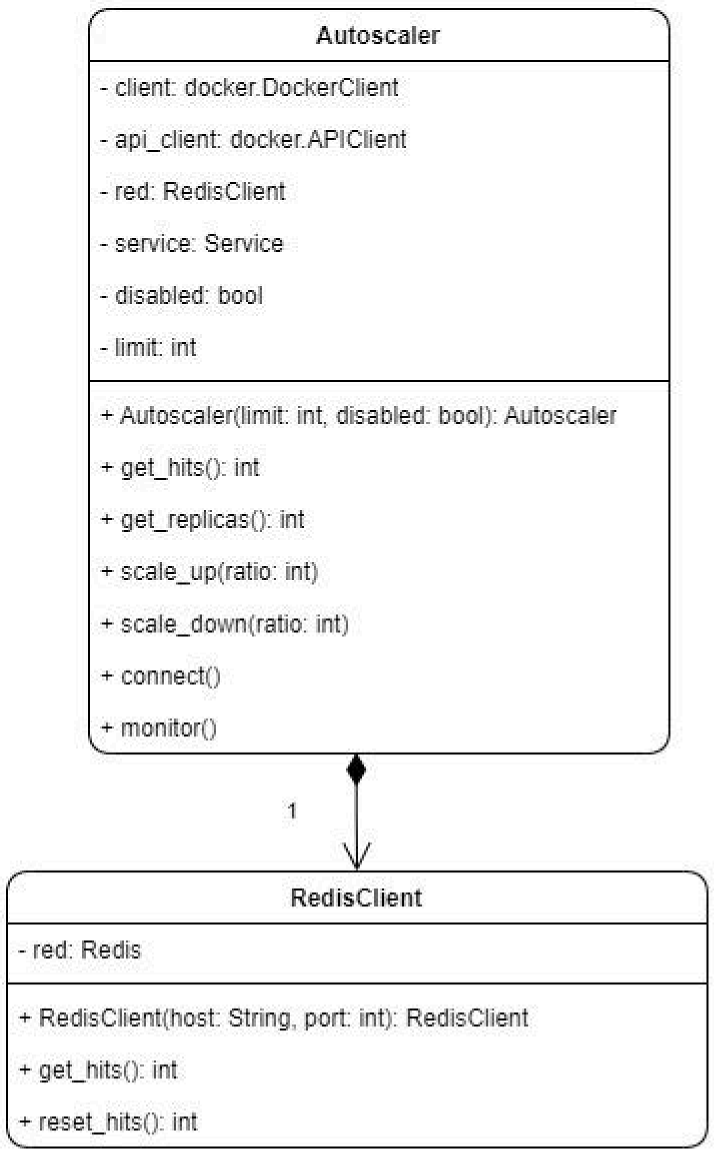
Cybera Rapid Access Cloud: Cybera is a non-profit infrastructure-as-a-service (IaaS) provider. Rapid Access Cloud is a cloud computing service that is free to use for University of Alberta students. Our project runs on three M1.small virtual machines, one of which will be the Docker Swarm manager, and the other running the workers.

Ubuntu: Our Cybera VMs are based on Ubuntu 20.04. This distribution of Ubuntu executes on Linux kernel 5.4, and is under long-term support.

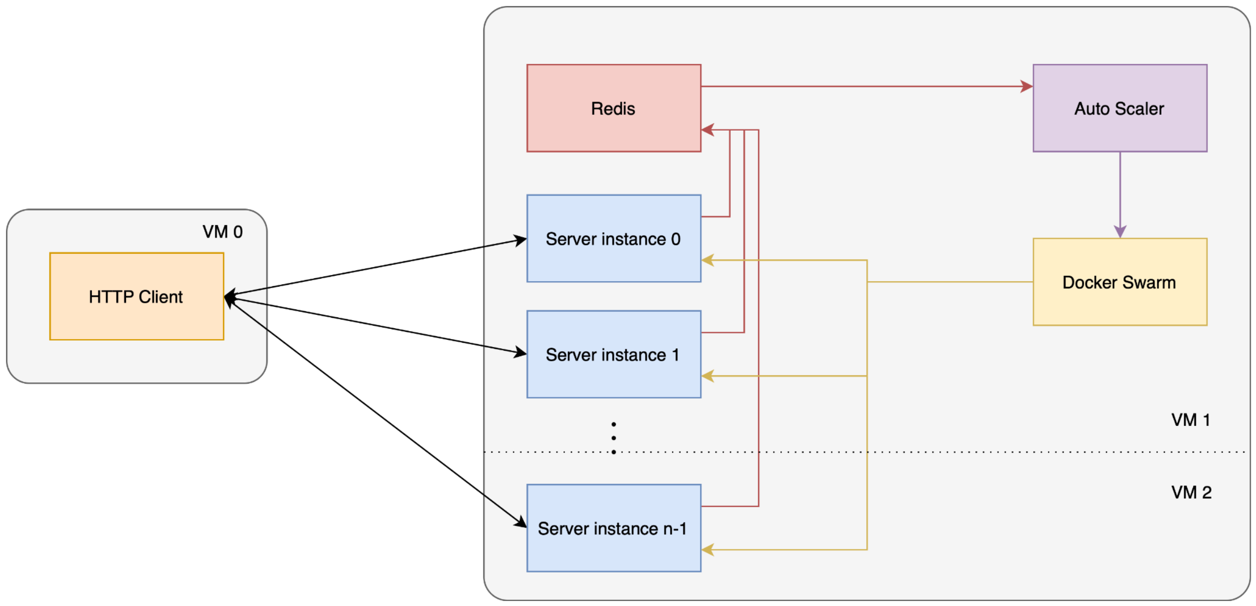
Python: Our auto-scaler will be written in Python, using a combination of the Docker SDK for Python and system calls. Using Python allows for smoother and simpler implementation and integration of our autoscaler plugin.

Design Artifacts

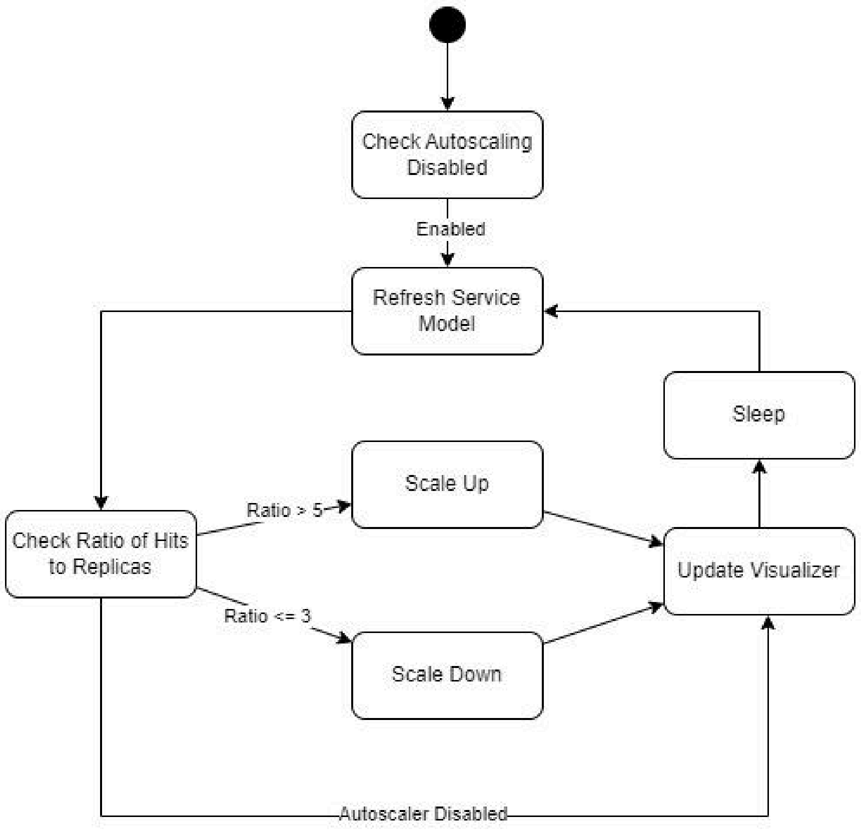
Class Diagram



High-level Architecture Diagram



State Diagram



Autoscaler Pseudocode:

```
Unset
// approx. ratio of hits per replica during last 10 seconds

while True:
     ratio = 2*requests / n_replicas
     if (n_replicas == 0) OR (ratio >= 5):
          scale up service by (1 + ratio - 5) replicas

     else if (ratio <= 3) AND (n_replicas > 1):
          // ratio can be very low, so we bound it
          bounded_ratio = max(ratio, 0.6)
          scale down by 3/bounded_ratio

// n_replicas is bounded between 1 and 50.
```

Deployment Instructions and User Guide

1. To begin with, follow the instructions listed in the ECE422-A1 starter code.
    a. Note: In step 6, replace *app_name* with "simpleweb".

Python setup:

1. On the Client VM:
    a. Move the directory (from the GitHub repository), *http_client*, to the Client VM.
    b. Move to the *http_client* directory.
    c. Run the command *'source setup.sh'.*

This will install python3.7, create a virtual environment, install all pypi requirements.

2. On VM 2:
    a. Move the directory (from the GitHub repository), *autoscaler*, to VM 2.
    b. Move to the *autoscaler* directory.
    c. Run the command *'source setup.sh'*

       See the above instructions for what setup.sh does.

Running the autoscaler:

1. Ensure that the Docker services from the docker-compose file are running.

```
Unset
$ sudo docker ps
```

   Look at the status column.

2. On VM 1, inside `http_client/`:

```
Unset
$ source env/bin/activate
$ sudo env/bin/python3 http_client.py \
     <IP_of_VM_2> \
     <num_of_users> \
     <sleep_time_between_requests_in_seconds>
```

3. On VM 2, inside `autoscaler/`:

```
$ source env/bin/activate
$ sudo env/bin/python3 autoscaler.py
```

To visualize service count and requests/sec:

1. Make sure that autoscaler.py is running on VM 2.
2. On VM 2, in the autoscaler folder:

```
Unset

$ source env/bin/activate
# run chmod +x run_tensorboard.sh if necessary
$ ./run_tensorboard.sh
```

And then Ctrl + click (or Cmd + click if you're on Mac) on the link: http://localhost:<port>/. If you are running this on bash, please port forward said port to your local computer.

Limitations

We observed that past a certain point, for a large number of users on our client (we tested this on up to 100 users), hardware limitations begin to minimize the benefits of scaling our service. We believe that this is due to the VMs in our Swarm only possessing 2 VCPUs each, which limits the benefits of multithreading past a certain threshold. In this case, the average time per request may increase rapidly.

Conclusion

This project demonstrates our implementation of an auxiliary service that dynamically scales up and down the number of web services depending on the volume of incoming requests. This is an important, small-scale application that performs the same basic functions as large-scale dynamic auto scalers implemented on SaaS / Cloud services such as AWS, Azure, and other cloud service providers. Overall, our autoscaler helps to ensure application performance and ensure users maintain a high quality of service.

References

[1] “Swarm mode overview,” Docker Documentation, https://docs.docker.com/engine/swarm/ (accessed Feb. 5, 2024).

[2] “Python guide: Redis-py,” Redis, https://redis.io/docs/connect/clients/python/ (accessed Feb. 5, 2024).