Reliability Project

Richmond Naviza and Isaak Coleman

Abstract

This paper documents the Reliability Project design, setup guide, user guide, and lessons learned. It documents the reasons for using Docker for scaling services, Redis for passing data from the client to the server, and Flask for serving as the web framework to present the collected data. The setup guide covers the required Python 3.8, Docker, Redis, and Flask setup. The conclusion contains discussions about software reliability, its relation to this project, how reliability must be balanced with cost, and how Docker is the perfect tool to accomplish scaling services.

Introduction

The purpose of this project was to learn how Docker works and gain experience by building a small Docker application. This project built off a starter application available here: Sohrabbeig / ECE422-Proj2-StartKit which implements a client/server application. The client sends requests repeatedly with a configurable delay, and the server performs some busy wait task before returning a response. When the wait time of the client is reduced or multiple clients are run the server will be unable to maintain an acceptable response time. This provides an opportunity for a scaling process.

Docker hosts applications in containers, which are hardware and operating system agnostic running environments. These containers can be duplicated to scale processes, such as our server. By monitoring the response times of clients we can actively scale the process up to meet demand. It is also important to down scale the process when additional replicas (duplicated containers) are not needed to save resources. In a real application this would save costs on server infrastructure.

Technologies, Methodologies, Tools

Docker

Docker is used to host the server and auto-scaling service in containers. The auto-scaling service duplicates the server container (increases the replica count) to reduce the response times of clients and decreases the replica count when it is no longer needed.

Redis

Redis is a RAM database that stores information using key-values. This makes it extremely fast and easy to use for small amounts of data. We extended the use of Redis to allow the clients to send their response times to the autoscaler. The autoscaler also uses Redis to store the number of replicas, average response times, and number of requests per cycle so they can be graphed later.

Flask (Python framework)

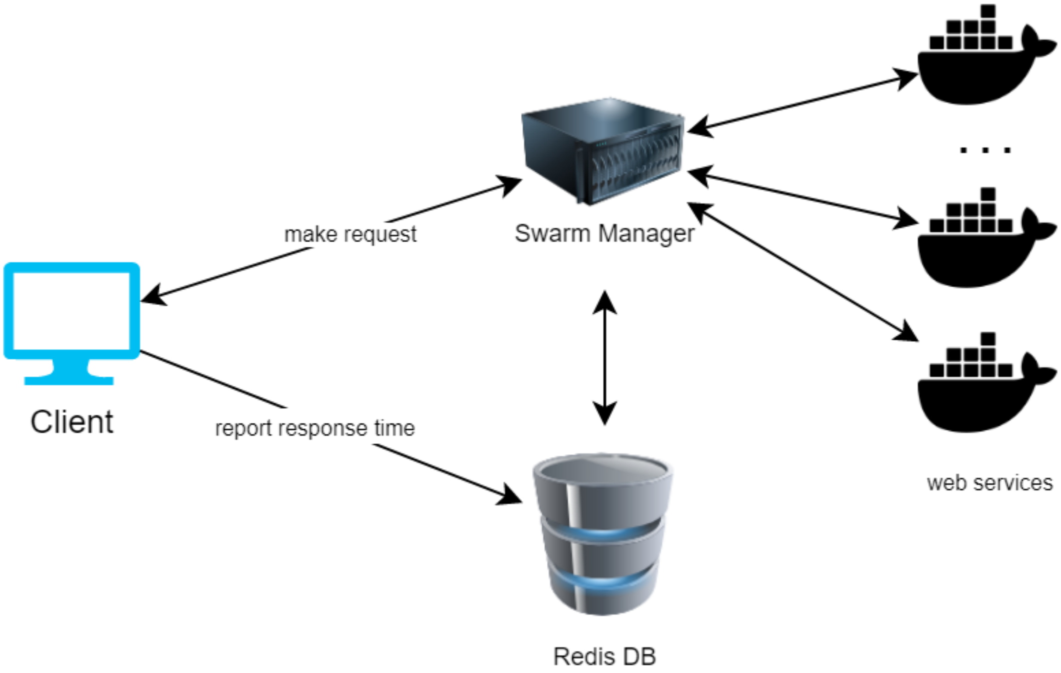
Flask is a Python web framework. In this project Flask was used by the starter kit to make web endpoints for the server. We extended the use of Flask in the autoscaler to allow viewing of the graphs mentioned above, and a toggle endpoint to enable/disable autoscaling.

Autoscaling Implementation

To get the response times of the server we added a section to the client program that pushes the response time to the Redis database. This allows the autoscaler to grab all of the response times and average them to make a scaling decision. We found the nominal response time of the server to be around two seconds, so we selected that as the lower bound. The response times could sometimes vary quite a bit, so we selected five seconds as the upper bound so that scaling could not be inadvertently triggered. We found the replicas would come online and reduce the response times in around ten seconds, so we made the autoscaler run in cycles of twenty seconds to get a good average and allow the change in replica count to significantly change the average before running again. We made the number of replicas increase by four when the upper threshold was passed to allow quickly scaling to higher loads and to help deal with backlog. We only decreased the replica count by one when under the minimum threshold to allow fine scaling. This can cause an overshoot of the optimal replica count, but makes up for it with fast load reaction and no rebounding.

Design Artifacts

High Level Architecture

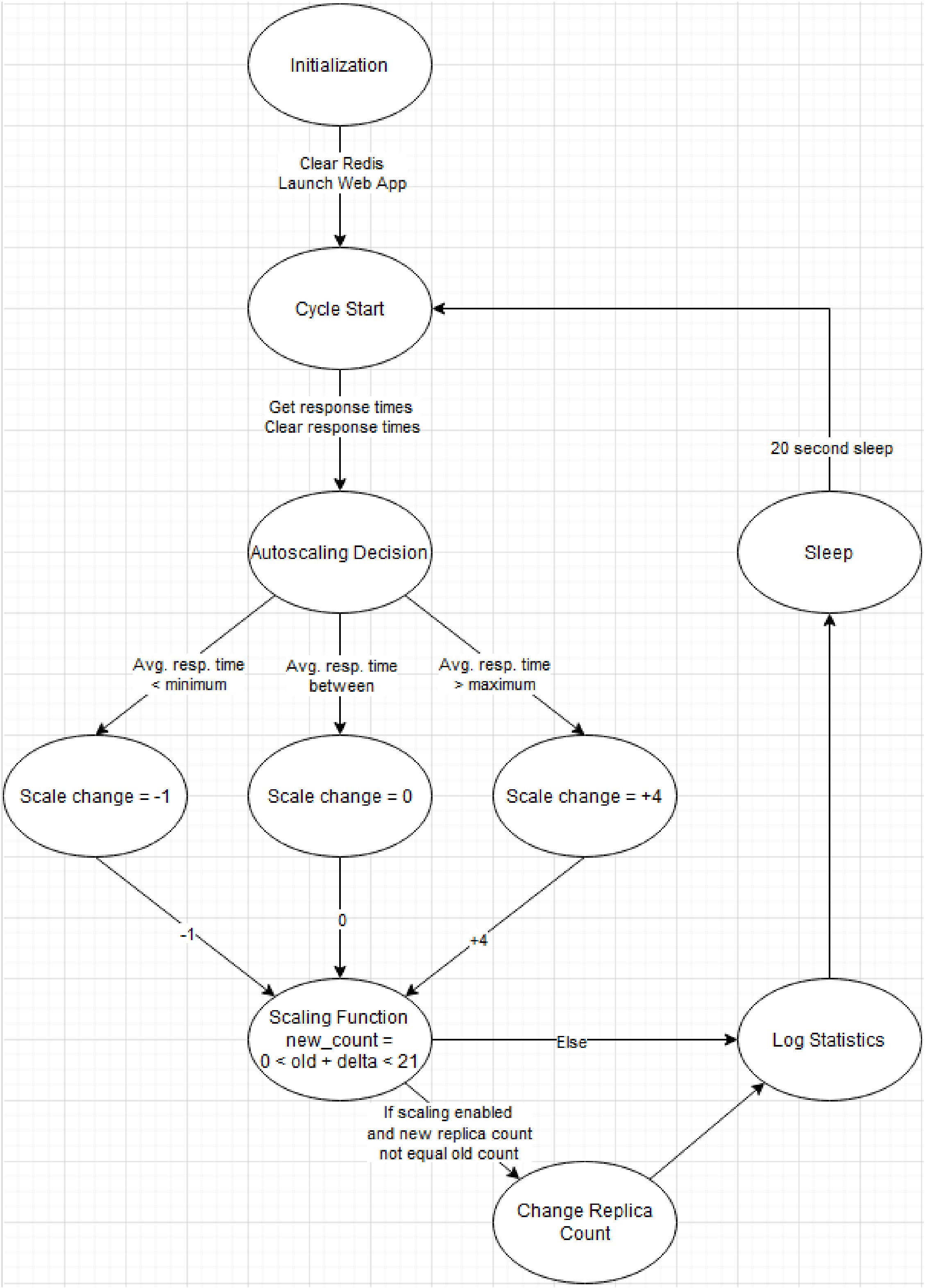


When a client makes a request, it accesses the simpleweb web service. This web service is managed by the swarm manager. After making (and finishing) the request, the client sends its response time to the Redis database. The autoscaler, which exists in the swarm manager VM, accesses the Redis database and aggregates the reported response times into an average value. If the average time is greater than the maximum acceptable time, the swarm manager will increase the number of web services. If the average time is below the minimum acceptable time, then the swarm manager decreases the number of web services.

Autoscaler Pseudo Code:

```
1. Get the list of reported response times from redis.
2. Clear the list
3. Take the average of the list
    a. If the average is lower than the minimum response time, decrease the number of containers by 1. The minimum container number is 1.
    b. If the average is higher than the maximum response time, increase the number of containers by 4. The maximum number of containers is 20.
4. Push the number of requests, the current number of containers, and the average response times to redis (to present the data)
5. Sleep for 20 seconds
6. loop back to step 1
```

Autoscaler State Machine



Deployment Instructions

To spawn 3 Virtual Machines and set up the docker services for the client and the swarm manager, follow the instructions given on the Reliability Project repository

For the Swarm Manager VM:

- Install Python version 3.8
- Install the following python packages:
  - redis
  - docker
  - flask

User Guide

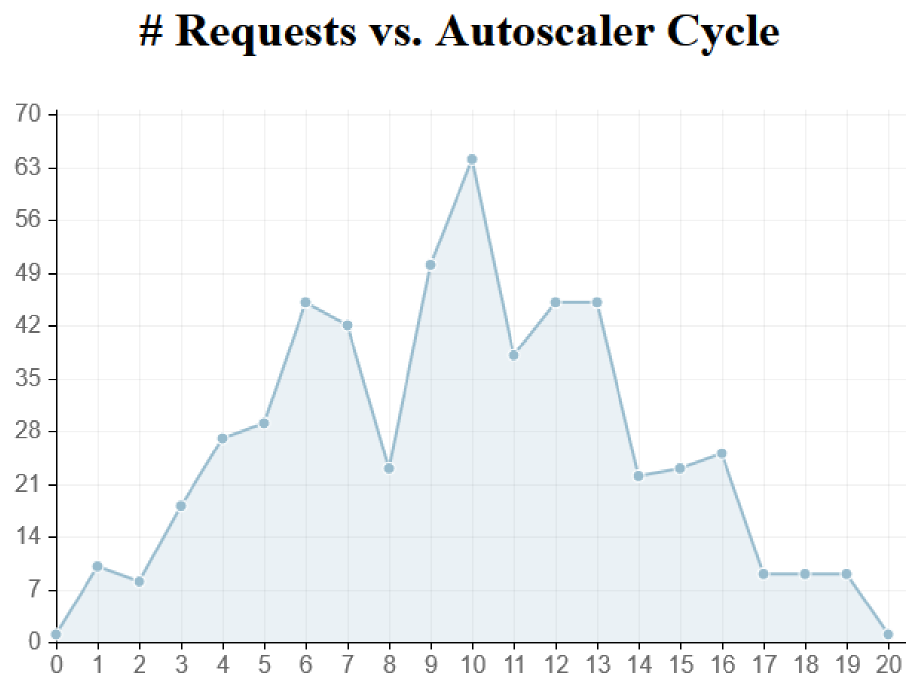
On the Swarm Manager VM:

1. Download the autoscaler folder from the Github repo to the swarm manager directory
2. Go into the autoscaler directory
3. Run the following command
   ```
   $ python3.8 autoscaler.py
   ```
4. Open a browser to localhost:1337 to view the graphs o
   a. Refresh the page to update the graphs
5. Go to localhost:1337/toggle to toggle the autoscaler on or off
6. Go to localhost:1337/clearlogs to clear the graphs

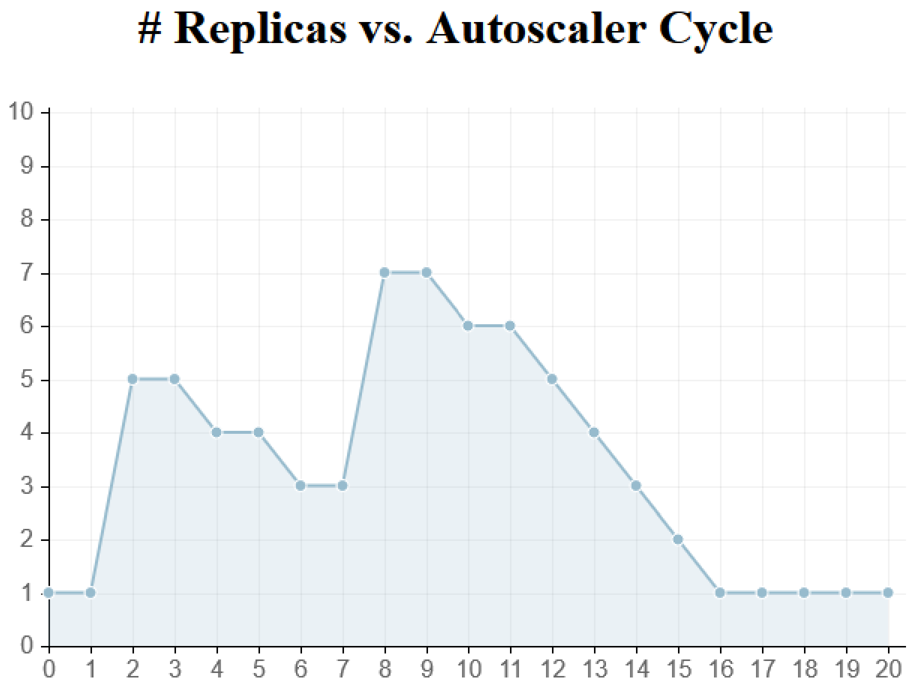
On the Client VM:

1. Make requests to the web service by running the following command:
   ```
   $ python3.8 http_client.py <swarm_manager_ip> <number_of_clients> <think_time>
   ```

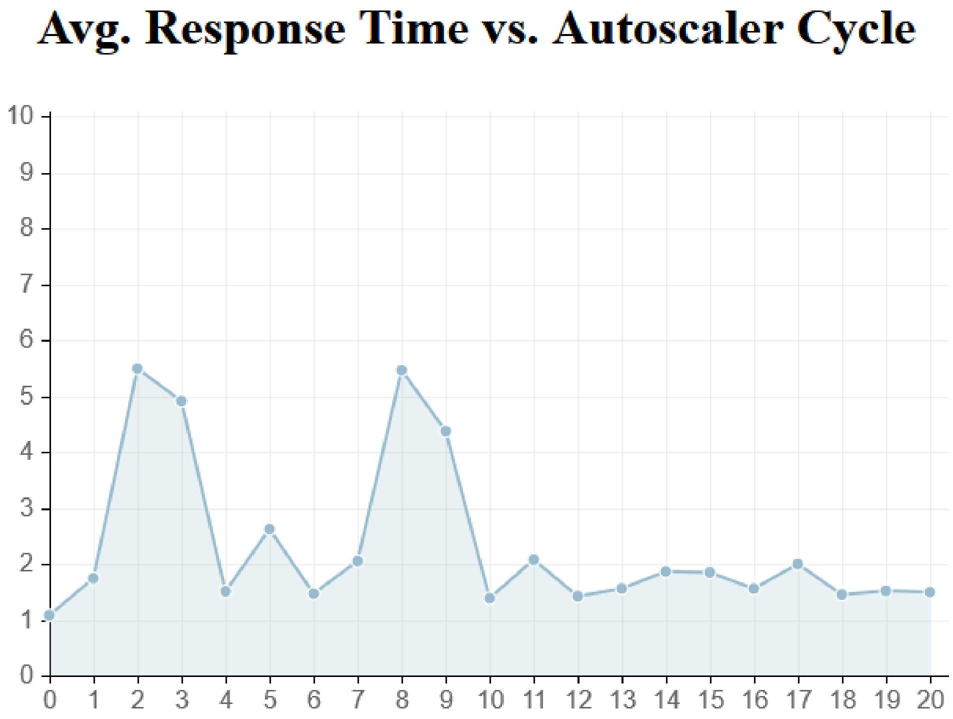
Graph Information



The requests per cycle graph shows the number of successfully returned requests in an autoscaling cycle. This can lag behind the number of actual requests if they start timing out. Because the clients have a set delay after receiving a response a faster server response time means a higher number of requests will be received per cycle.



The replica count graph simply shows the number of server replicas at the start of an autoscaler cycle. We increased the number of replicas by four when the response times went above the acceptable threshold, and decreased by one when the response times fell below a minimum boundary.



The average response time graph shows the average of the response times received from the client over the autoscaler cycle. Comparing the average response time and the change in the container count graph at the same autoscaler cycle shows when it decided to up/down scale.

Conclusion

Software reliability is a multifaceted endeavor and has many aspects including the ability of software to perform its required functions at an acceptable level for a specific period of time. In this project, we were tasked with creating a system that allowed for a server to respond to requests within a specific acceptable window of time. This mirrors a real world application where a client must be serviced within a short time period but also balance the computing resources on the server amongst running services and minimize the operational cost on the cloud.

A Dockerized web service is the perfect solution to handle an autoscaling project since it allows for replicating a service in order to optimize performance and delete replicas of services in order to optimize for cost by only using resources when necessary.

References

A. Iannino and J. D. Musa, “Software reliability,” *Advances in Computers*, 30-May-2008. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0065245808602995. [Accessed: 15-Apr-2022].

A. Sohrabbeig, “Sohrabbeig/ECE422-proj2-StartKit,” *GitHub*. [Online]. Available: https://github.com/Sohrabbeig/ECE422-Proj2-StartKit. [Accessed: 15-Apr-2022].

“Docker documentation,” *Docker Documentation*, 14-Apr-2022. [Online]. Available: https://docs.docker.com/. [Accessed: 15-Apr-2022].

“How to set chartjs y axis title?,” *Stack Overflow*, 01-May-1963. [Online]. Available: https://stackoverflow.com/questions/31913967/how-to-set-chartjs-y-axis-title. [Accessed: 15-Apr-2022].

Ruan, “Graphing pretty charts with python flask and Chartjs,” *Graphing Pretty Charts With Python Flask and Chartjs*, 14-Dec-2017. [Online]. Available: https://blog.ruanbekker.com/blog/2017/12/14/graphing-pretty-charts-with-python-flask-and-chartjs/. [Accessed: 15-Apr-2022].

“Software engineering software reliability - javatpoint,” *www.javatpoint.com*. [Online]. Available: https://www.javatpoint.com/software-engineering-software-reliability. [Accessed: 15-Apr-2022].

To Do:

- Proofread and Edit
- Upload things to Github
- Retest the autoscaler a couple more times