ECE 422 Project 1

Prepared By Team Goon (Justin Javier, Saif Husnain)

GitHub: [2Bronze/ECE422-Proj2-StartKit (github.com)]

Abstract

This report details the design, development, and progress of an auto-scaling system that attempts to optimize performance and cost of a cloud-based web application. The implementation utilizes Docker containers for deploying microservices for flexibility. Response times of requests are monitored as they are completed and graphed. The auto-scaler monitors if the response time is acceptable by placing upper and lower bounds on an acceptable scale. If the upper bound is exceeded, the system scales out (increases the number of replicas within docker) and if the lower bound is past, then it scales down accordingly. The project shows the importance of intelligent auto-scaling mechanisms in cloud computing due to the possibilities of greater scalability, efficiency, and reliability.

Tools and Technologies

Docker: Docker was chosen because it allows us to create new instances of an image. This means we can start up new instances of an application without worrying about the variables for operations (i.e. OS, required environment variables, installed software, etc.). In addition, it is lower cost than running a VM.

Python3: Choose python for the implementation of the auto-balancer and auto-scaler. Python was chosen because of its high-level language capabilities and our group's programming experience in python.

Cybera: Hosting the applications. Cybera is a cloud resource management service that allows us to start up VMs to host our application on. Cybera was chosen because it was mandated in ECE 422.

Agile: Used to plan the development of the project. This was used due its iterative nature. Our current iteration for implementation is currently limited by our inexperience in creating a similar product. Agile allows us to amend and change plans as we encounter issues.

Burndown Chart: Tracks the progress of the project according to tasks. Allows us to visualize if we are on track to complete the project before the deadline.

Design Artifacts

Original Class Diagram:

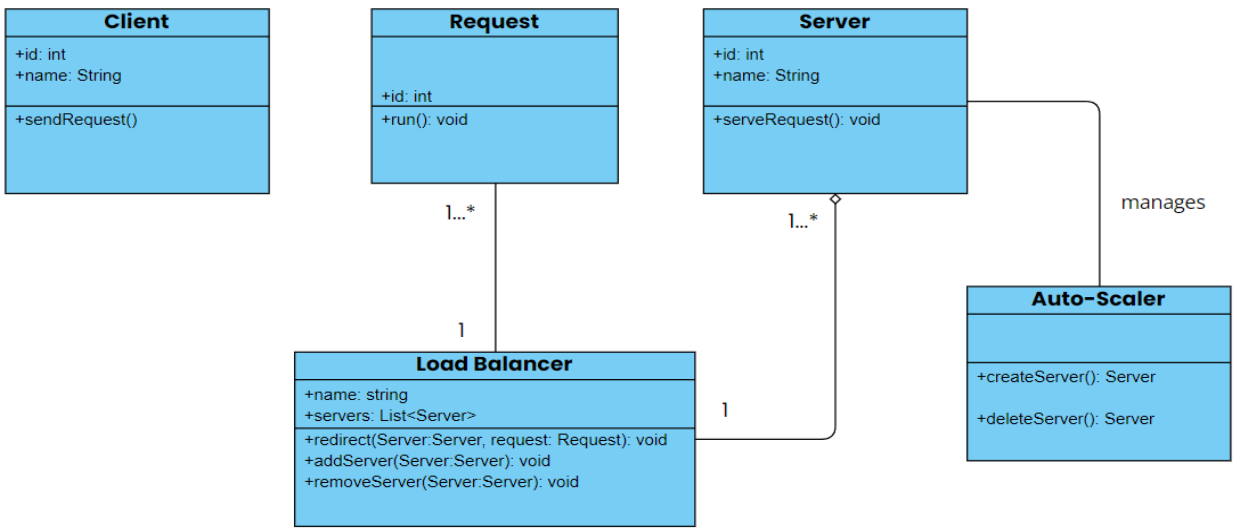

Figure 1: Class Diagram

The original class diagram that was created before beginning development on the project. This was presented in the original deliverables. Once beginning the project and development, we realized that a few things needed to be slightly adjusted. While the overall workflow is similar, to note a few of the changes:

- There is no Request class specifically created, however pinging the server fulfills the purpose
- Load Balancing is done by docker automatically, so it did not need to be explicitly implemented.
- The one class explicitly stated was the Scalar and includes the functionality listed. The methods were changed to upScale() and downScale() since they more accurately describe what is going on.

Updated Class Diagram:

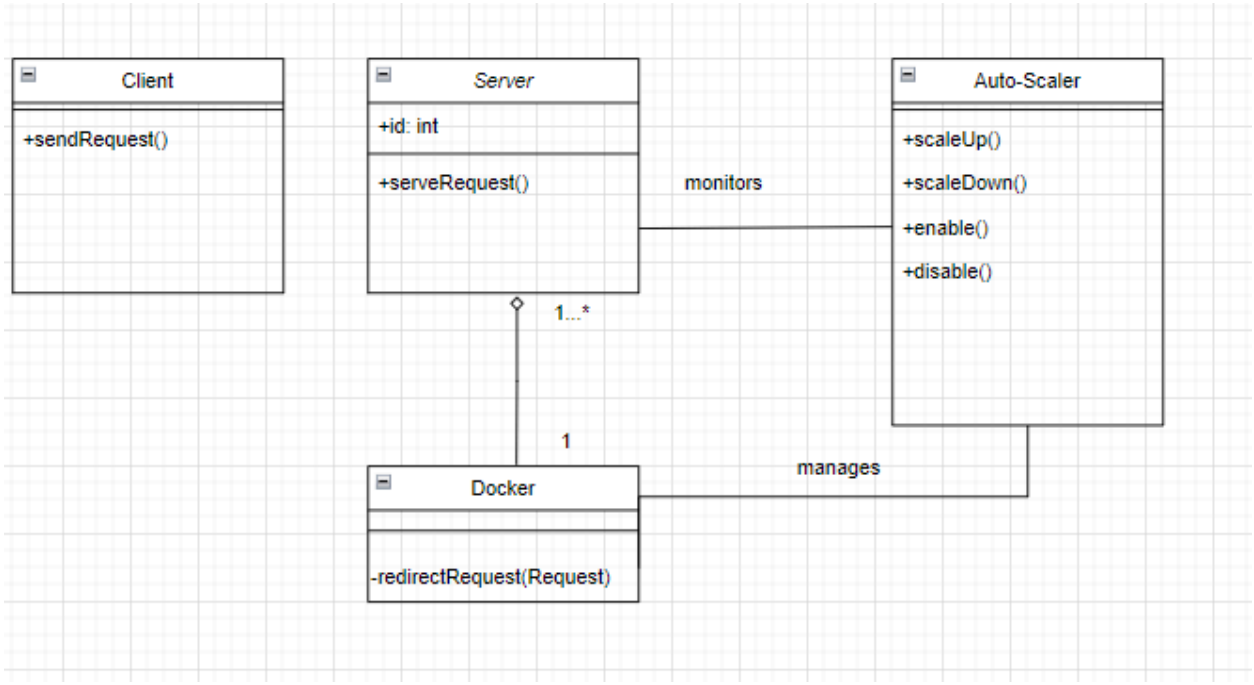

Figure 2: Updated Class Diagram

The updated diagram is more accurate to the actual system. In actuality the auto-scaler monitors the server for response times, but creates replicas through Docker. There are also methods within the scalar to enable or disable scaling.

State Diagram:

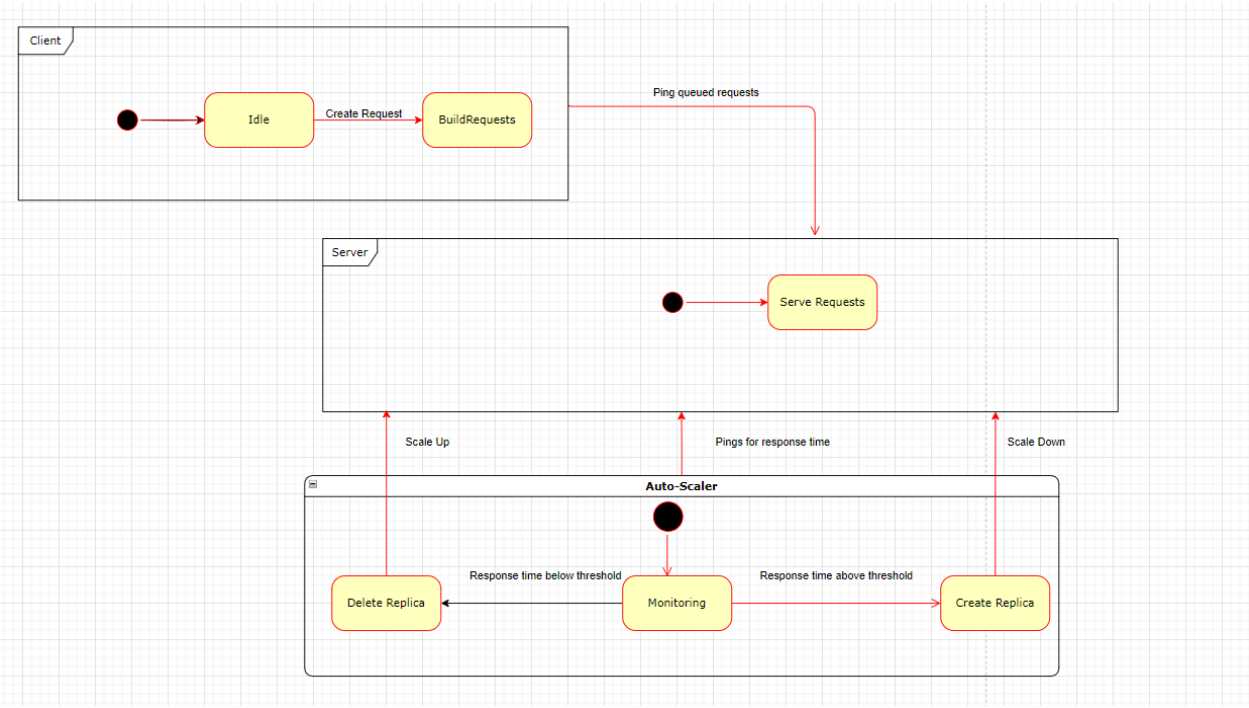

Figure 3: State Diagram

The above state diagram describes the general workflow. The client builds requests and queues them up. When it is ready to ping the server, it does and the server begins serving the requests. While this is happening the auto-scaler monitors the server and when a threshold is crossed either way, it either creates or destroys replicas accordingly. The system requires to be started, but is continuous, therefore there are no end states.

Original Timeline:

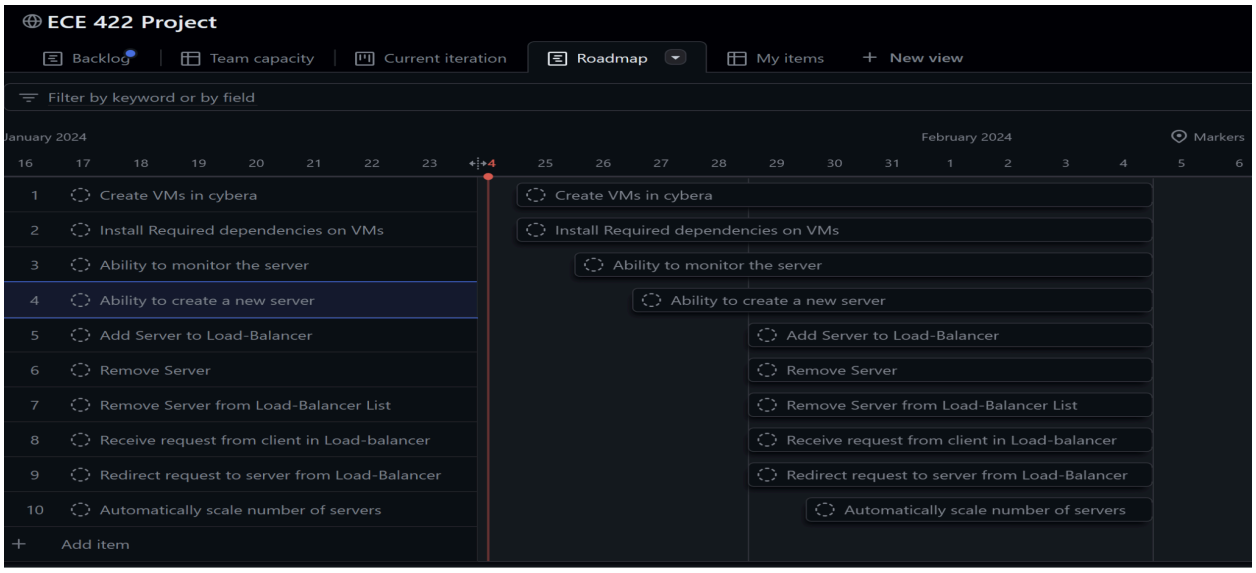

Figure 4: Original Goal Timeline

The timeline was the original one created for when the tasks were going to begin and around when they would end. The final timeline is better represented using the burndown chart.

Burndown Chart:

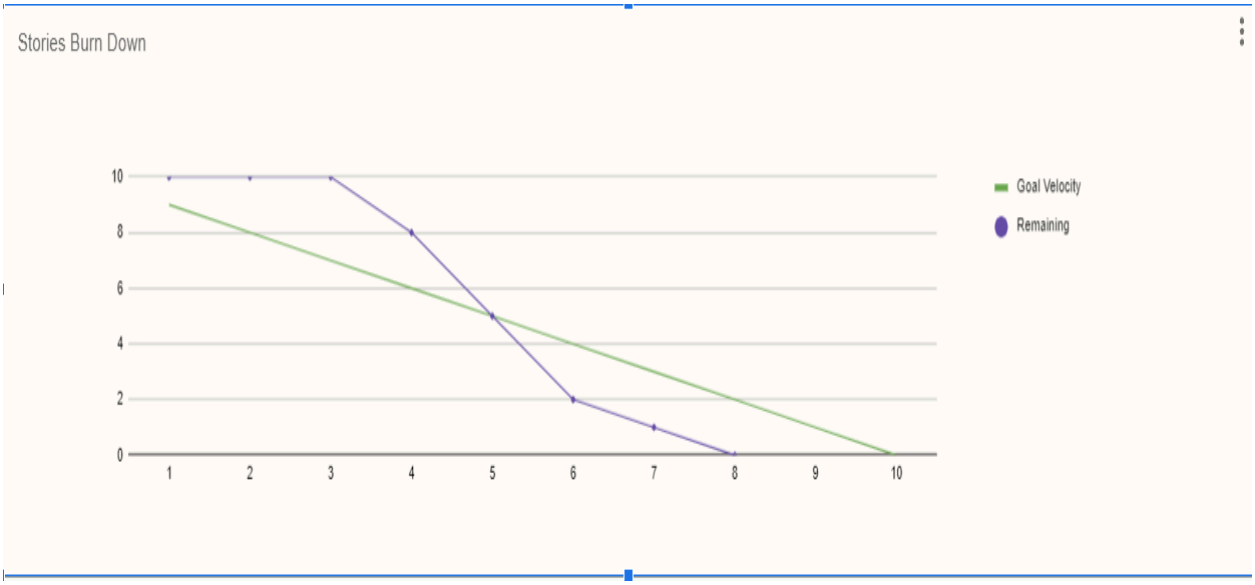

Figure 5: Burndown Chart

The burndown chart above shows the progress and timeline that was achieved. To explain the timeline, over a 10 day period we outlined how many user stories shall remain. The remaining line in contrast is what the actual timeline ended up being. Overall, it took a little longer to begin than anticipated, but we were able to finish user stories faster than originally expected.

Performance Charts:

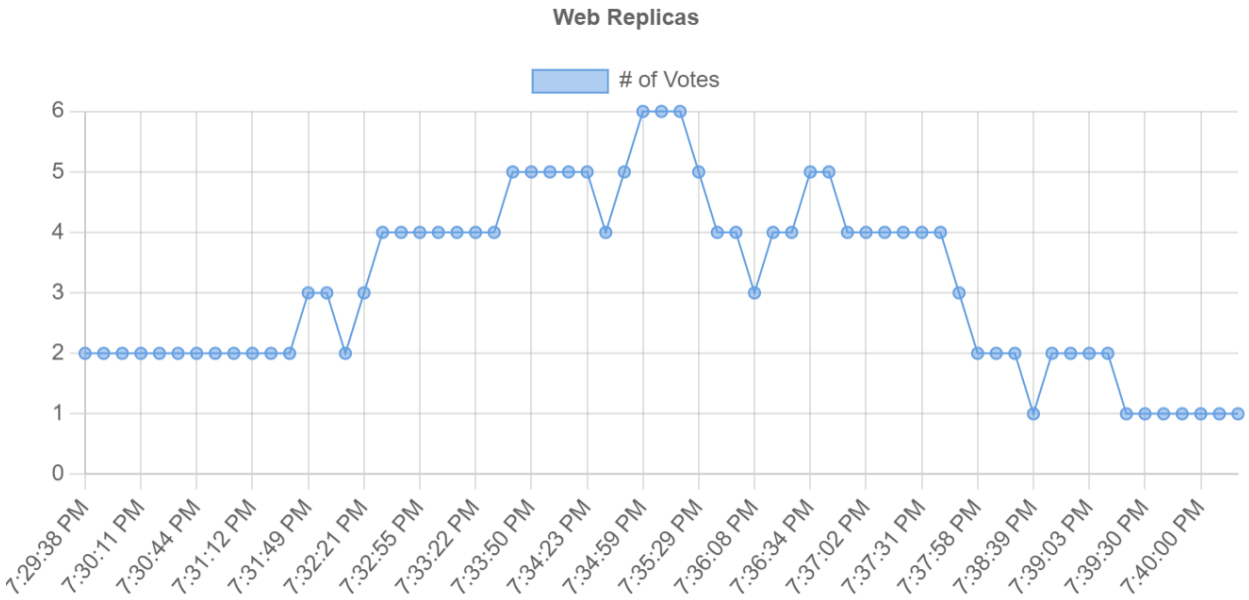

Figure 6: Replicas Chart

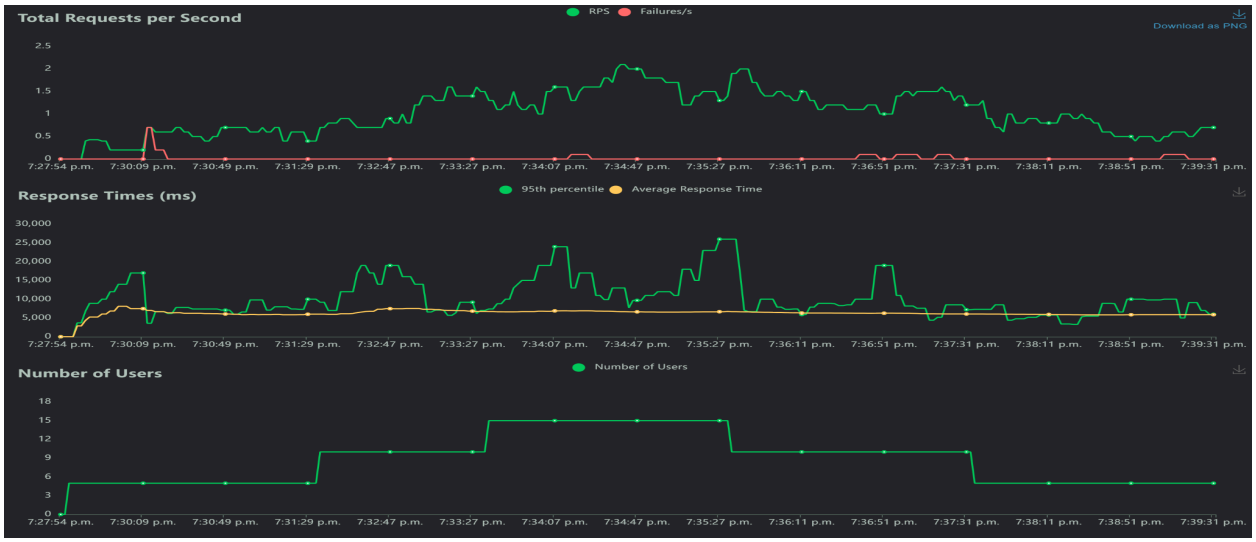

Figure 7: Performance Charts

Here the performance charts show us as the number of requests increase per a normal distribution, we can see that as the replicas increase the response times decrease, before the requests increase past the upper threshold again, where we see a similar pattern occur of response times going up and then back down once the upscaling occurs.

Deployment Instructions

1. *ssh* into the docker swarm master VM
2. Clone the repository
   git clone https://github.com/2Bronze/ECE422-Proj2-StartKit.git
3. Navigate to *./docker-images/auto-scaler*
4. Build the image
   sudo docker build -t scaler:latest .
5. Navigate to the repository root
6. Deploy via docker-compose file
   sudo docker stack deploy --compose-file docker-compose.yml app_name

The following services will be provided at your swarm's IP address:

- Port 4444: Auto-Scaler UI
- Port 5000: Docker Visualizer
- Port 8000: Web Application
- Port 8089: Locust

User Guide (Auto-Scaler UI):

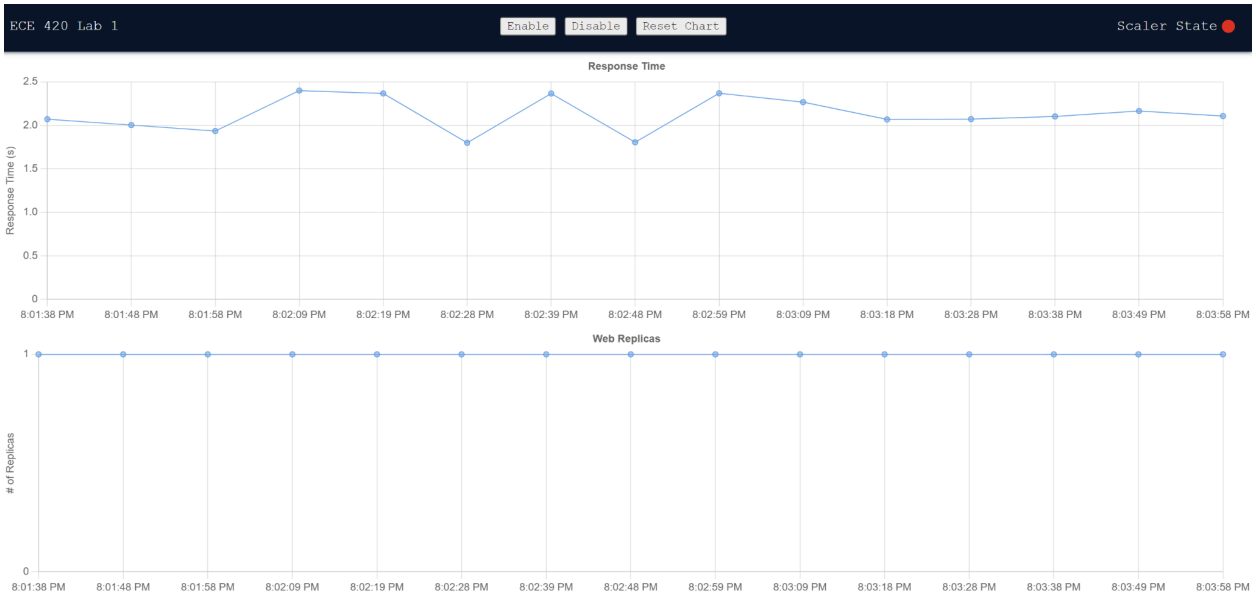

Figure 8: Auto-Scaler UI

The auto-scaler UI consists of 3 buttons and 2 graphs. The enable button will enable the auto-scaler. The disable button will disable the auto-scaler. The auto-scaler's status can be seen on the top right corner of the UI. The graphs will update every 10 seconds with

the latest data of the web microservice’s response time and the number of web replicas currently running.

Conclusion

Having created an auto-scalar to create replicas of the original server to distribute the load, it is found that a well implemented scaling system improves scalability greatly. Having created a normal distribution of requests per second over a 10 minute period, we find that the behavior is as expected. While the requests go up and the scaling doesn't kick in, we see an increase in response times, however once scaling occurs, a decrease in response time follows.

References

[1] “Docker SDK for Python,” Docker SDK for Python - Docker SDK for Python 7.0.0 documentation, https://docker-py.readthedocs.io/en/stable/ (accessed Feb. 9, 2024).