ECE 422 - Project 1 Final Report

TeamCBC:
Brandon Hoynick
Calvin Choi
Carl Fang

February 9, 2024

Abstract

Because of the nature of client-server interactions, the ability for one’s services to be consistent regardless of incoming workload is paramount. Ideally, clients should see consistency on their ends, be it uptime or responsiveness. In consideration of software reliability, this project develops a solution for automatic scaling of services when faced with various workloads from incoming clients of varying intensity.

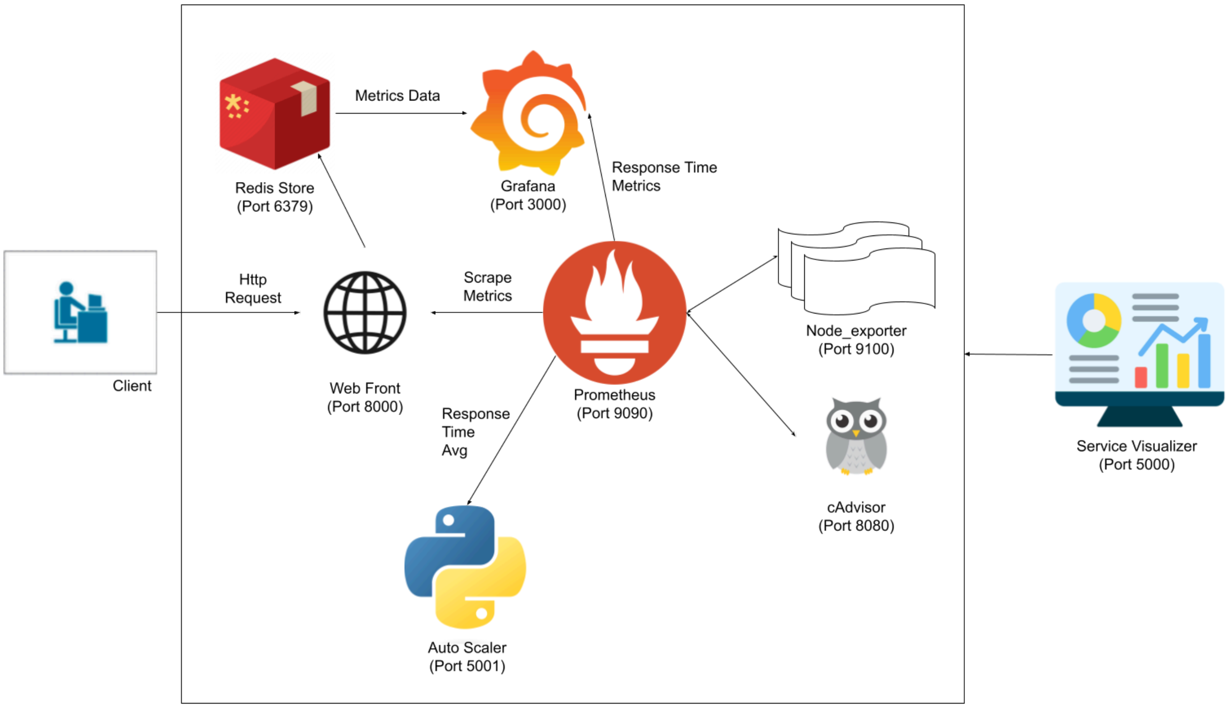
Docker is like a virtual application deployer, which can properly handle multiple instances of applications to suit the multiple instances of clientele. In our project, we use Docker, along with python scripts, and supplementary services to create a scalable web app server.

Technologies/Methodologies

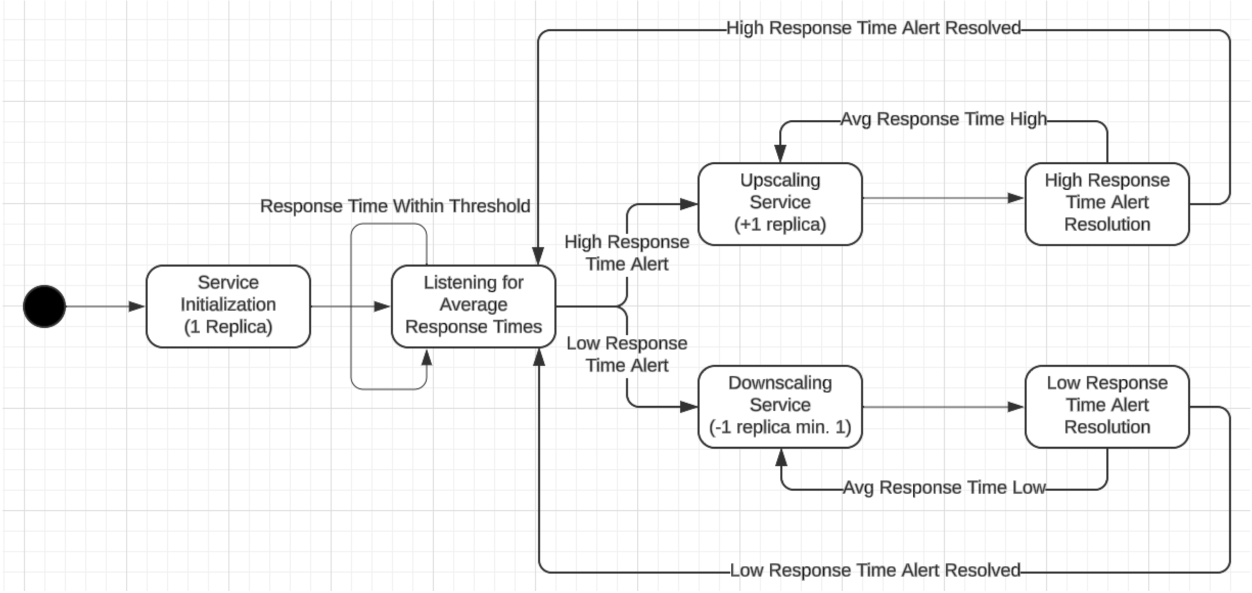
Technology	Use case/Explanation
Cybera	Cybera was suggested for all of this project's cloud services needs. In particular, this works well in that its servers are local (Alberta) and thus, easily accessible as well. It provides all the necessary cloud services for this project to function at base.
Python	Python is the main language of choice for the project. It was selected for its ease of use and flexibility for whatever it was needed for, be it the web app or data scaler.
Grafana	Grafana is the main tool for data visualization. It was selected due to ease of integration and powerful visualization tools over time. Importing data is a simple flow, and can be consumed and displayed with ease with Grafana.
Redis	Redis is the databasing solution used in the project. It was used for its simplicity of integration, and because there was a lack of a need for a db of greater substance, given that we didn't store much anyways.
Flask	Flask is the web framework used for the web application in the project. It was selected for its simplicity, given that it's a lightweight framework compared to something like React. The web app itself didn't need much complexity in the framework.
Prometheus	Prometheus was used for the project's monitoring and alerting needs. It was selected for smooth integration with docker, and the ability to track the export of the specific metrics we need easily with tools such as node_exporter cAdvisor. (It also has nice easy integration with Grafana).
Docker (and docker-compose)	Docker was the main tool used for containerization and scaling of the services in the project. It was used for its easy command line scaling abilities and ability to manage workers easily.

Design

High-Level Architecture Diagram



Auto-Scaler State Diagram



Auto-Scaling Algorithm Pseudocode

The actual high-level algorithm for the auto scaler is relatively simple. It essentially boils down to:
(Given upper threshold **X** and lower threshold **Y**)

```
if response_time > X:
        scale_up()
else if response_time < Y:
        scale_down()
```

As for the scale_up() and scale_down() functionality, the logic is simple as well. Upwards scaling increments the number of replicas by one:

```
def scale_up():
        replicas++
```

As for downward scaling, it decrements the replicas by one, but maintains a replica minimum of 1 existing replica.

```
def scale_down():
        if (replicas > 1):
                replicas--
```

In order to implement a more robust scaling policy, this logic is handled by our Prometheus alert manager. When abnormally high or low average response times are detected, an alert is fired off and handled by the manager. After scaling up or down, the alert state is maintained until conditions to escape it are met, thus scaling upwards and downwards faster when conditions are far from being met.

As for the specific parameters we chose, we measured average response time in seconds. We took a lower threshold of 1 seconds and an upper threshold of 3 seconds. This was based mainly on manual testing for varying client workloads providing average response times from the given client that processes a difficult function. The bounds were also selected to be 2 seconds apart, as to compensate for the natural hills and troughs of a stable set of response times. We found through testing that this was a comfortable threshold gap. The average monitoring interval (refresh rate) is 15s.

Note about our Github repo page

- Our server side elements are located within the ‘Project1’ folder (which contains all microservice help apps for the web app serving system).
- The ‘original-kit’ folder contains items from the starter kit (not used, just kept for the original reference).
- The ‘ClientVM’ folder contains client simulator elements used in our project.
- The ‘Design’ Folder contains graphics and pseudo code seen within this report.

Deployment Instructions (and any user guides)

Note: You may require prefixing most commands with “sudo” depending on if the admin user is logged in.

1. Assure that the auto-scaler image is up to date. The image can be built and pushed by running the build.sh file in the auto-scaler directory.
    a. Run **./build.sh** to build and push the updated image for the auto scaler.
2. Assure the web-app image is up to date. The image can be built and pushed by running the build.sh file in the web-app directory.
    a. Run **./build.sh** to build and push the updated image for the web-app.
3. In order to deploy all the various microservices, run the following:
    docker stack deploy --compose-file docker-compose.yml app_name
4. Initialize any number of client connections to the running server application, and explore all the various microservices. This could be with the original http_client.py:
    python3 http_client.py <no users> <wait time>
    Or with Locust:
    locust -f locustfile.py
    (locust prompts to open http://localhost:8089 to see the Locust dashboard, where you can start the swarm and view the “Total Request per second”, the “Response Times (ms)”, and “Number of Users”).
5. View on various monitoring systems the different available charts (Locust, Grafana on localhost:3000 for retrieving Redis database data, **docker stats**, and Prometheus web scraper).
6. You should observe the autoscaler receive web statistical data which will check if response times are too high and tell Docker to scale up the amount of apps, or that response times are dropping to normal rates, so after a period tell Docker to scale down the amount of web apps.

Conclusion

Overall, the importance of auto scalability for a set of microservices is proven to be paramount when faced with dynamic workloads. As exemplified by the bell-curve shaped client request pattern, the workload that an application can face can shift dynamically very quickly. Scaling the app’s services manually proves to be inefficient and incapable of keeping up. Thus, to assure the overall software reliability of the application in the face of dynamic workloads, the task of application scaling should be left to an automatic scaler.

References

- Image References:
  https://eclass.srv.ualberta.ca/mod/resource/view.php?id=7494679
- https://www.vecteezy.com/vector-art/26221538-internet-icon-vector-symbol-design-illustration
- https://www.iconfinder.com/icons/202809/redis_icon
- https://en.wikipedia.org/wiki/File:Python-logo-notext.svg
  https://en.wikipedia.org/wiki/File:Grafana_icon.svg
- https://iconduck.com/icons/27828/prometheus
  https://www.netdata.cloud/integrations/data-collection/containers-and-vms/cadvisor/
- https://www.flaticon.com/free-icon/data-visualization_10397103

Docs and Help:

- https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/
- https://www.innoq.com/en/blog/2020/04/scraping-docker-swarm-service-instances-with-prometheus/
- https://prometheus.io/docs/introduction/overview/
- https://chat.openai.com
- https://redis.io/docs/get-started/
- https://grafana.com/grafana/
- https://locust.io/
- https://docs.locust.io/en/stable/custom-load-shape.html
- https://github.com/locustio/locust/tree/master/examples/custom_shape