ECE 422 Project 1 Final Report

February 9, 2024

Student Name	ID
Francis Sepnio Jr.	1617559
Defrim Binakaj	1621841
Reynel Guerra-Pavez	1616041

Abstract

This project presents an autoscaling microservice for Dockerized web applications based on the response. We used Docker's services to dynamically adjust the amount of container replicas based on the observed response times. Using a Python-based implementation, response time data is collected periodically, and based on that data the autoscaler will make decisions according to the predefined thresholds and scaling factors. We used Locust to test and measure the effectiveness and efficiency of our autoscaler by simulating user workload and generating various plots. The autoscaler successfully demonstrated its ability to adapt to the changing workloads while maintaining its reliability. This project offered hands-on experience with containerized application management, which demonstrates a practical solution to ensuring performance and scalability with different workloads.

Introduction

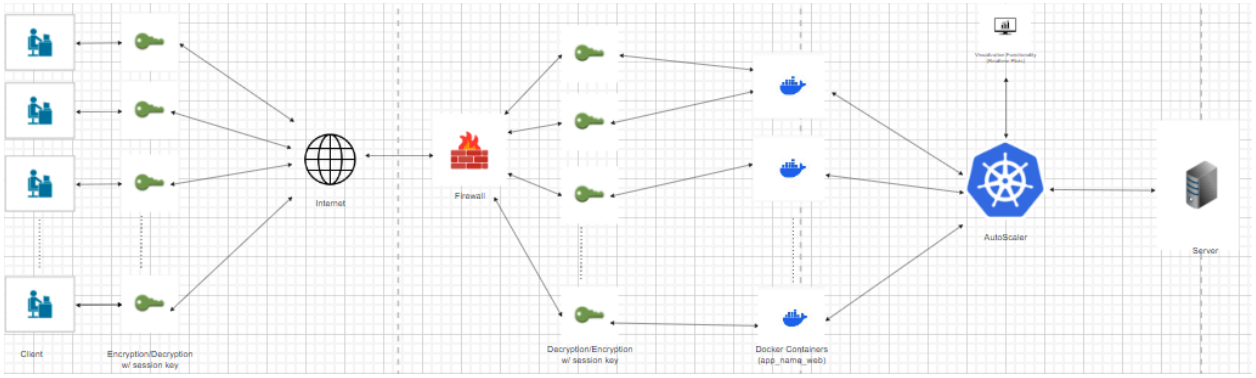
As the accessing of web servers has become a daily necessity, it is important for professionals like system administrators and site reliability engineers to monitor the performance of these systems to ensure that they are stable and optimal. When too many users are accessing a web service, the response time for the users' requests often increases as there are inadequate resources to support such a workload. When too few users access a web service, there are excess resources being wasted, which can cost companies lots of money. This project aims to create an auto-scaler program which is able to optimize performance and operational costs, by increasing the resources (via containers) of a web service when the usage is high, and decreasing the resources when the usage is low.

Technologies/Methodologies/Tools

Autoscaling of a web application through container management using Docker is tried and tested with Python – they have a specific Python Docker SDK for this exact use case allowing for powerful and efficient programming. Using **Python** with libraries such as **requests** for HTTP calling and **time** and **datetime** for time tracking, we were able to create a robust autoscaling program that utilized the **Docker SDK** for container scaling and management. **Pyplot**, from Matplotlib, was used to plot a graph of the number of containers over time (in other words, application size). When deciding on what platform to use for our real-time performance graphs, we decided on **locust**, since it provides plots of container information and performance indications, and also allows us to manipulate the load on our system in order to test various use cases and situations. This meant that our program consisted of an autoscaler python file for the scaling functionality, and a locust python file which was used to set our desired system-load-increase intensity and duration, afterwards directing the user to a webapp (on **browser**) that displayed all of the metrics and information.**GitHub Projects** was used to monitor the project's progress using its ticket system. **Git** was used for version control because of simultaneous development, allowing us to easily fix merge conflicts and finish the program.

Design Artifacts with Explanation

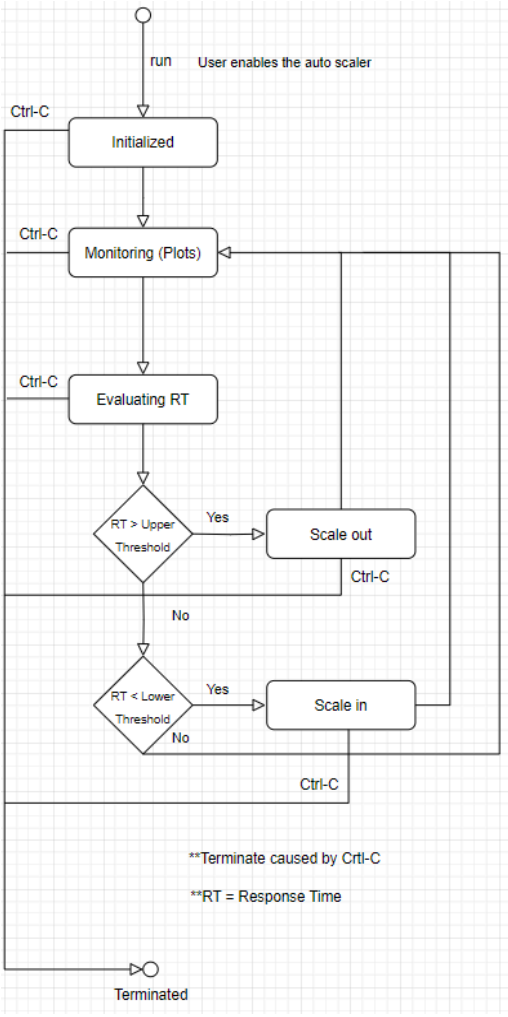
High-level architectural view



The High Level architecture diagram represents the flow from a client all the way to our server. Client will send a request, and the request will go through with the proper security measure. Need to be on the VPN, and sessions need Cybera Key-Pairs (specifically for our group since we are all on windows, using the VPN to access Cybera Instances). Once the request goes through, the Client will be pinging one of the Docker containers replicas. Our auto-scaler will take care of the amount of replicas that will be deployed, it will adjust based on the average response time, which we get from the server itself. At the same time, the autoscaler will implement real-time monitoring by plotting the request metrics.

State diagram

The state diagram illustrates the behavior of the autoscaler across different states. Firstly, the user must enable the Autoscaler. The autoscaler will start monitoring the requests metrics, plotting them in real time. Once requests start coming in the autoscaler will then evaluate the response time by contacting the server. We then check if the response time surpasses our upper limit threshold hold. If so, we then will scale out, deploying more containers. If not, check if the response time is lower than our lower limit. If so, we then will scale in, reduce the amount of containers deployed. After scaling out, scaling in, or doing neither, we return to the monitoring state (the start of the cycle). At any stage if termination happens (ctrl-C) then the program will terminate.



Pseudocode of the Auto-Scaling Algorithm:

```
While True:
|    While an interval of 20 seconds are elapsing :
|    |    Total response time = 0
|    |    Number of samples = 0
|    |    Get response time from server
|    |    Add response time to a total response time variable
|    |    Increment number of samples
|
|    If the number of samples obtained is nonzero:
|    |    Calculate average response time by dividing total
|    |    response time by number of samples
|    |
|    |    If average response time > 3 seconds:
|    |    |    Scale service out by 2x
|    |    |    Wait until containers reach expected value
|    |
|    |    If average response time < 2 seconds:
|    |    |    Scale service in by 0.5x
|    |    |    Wait until containers reach expected value
```

Reasonings behind parameter settings:

- A 20 second monitor interval was chosen because it can take some time for the containers to become stable and we wanted to reduce oscillation to the greatest degree. We want to ensure that we are getting sufficient average response time data before deciding if we need to perform any scaling on our application.

- An upper response time threshold of 3 seconds and a lower threshold of 2 seconds was chosen mostly through trial and error and observing the average response time of a server under medium load. When the average response time is greater than 3 seconds then that's usually a sign that we might need more

containers, whereas if the response time is less than 2 seconds then that is a sign that resources may be wasted and we should scale in.

- The scaling factors of 2x and 0.5x were also chosen through trial and error. We chose these scaling factors because we wanted to keep it simple and we found that we obtained good visualization results with these scaling factors.

Deployment Instructions and User Guides

For the purposes of the deployment instructions, it is assumed that it is known how to SSH into the swarm manager. Two instances of the swarm manager need to be running, ideally in an IDE such as VSCode.

Firstly, to open the output of the visualization microservice, connect to your swarm manager IP with port 5000. Here you will be able to see all of the containers “app_name_web”. As the autoscaler increases and decreases the number of docker containers, you will be able to see them visually.

In order to run the autoscaler, you want to ensure that Python is installed on your system and that you are using a Unix system.To install all of the dependencies:

1. Run `git clone https://github.com/PerrieFanClub-ECE422/Project-1.git`
2. Cd into the Project-1 folder
3. Run `pip3 install -r requirements.txt`.

Next step is to open Locust, which will simulate multiple client users to send requests to our web server. In one of the two swarm manager instances, run `locust -f locustfile.py`. This will open a Locust instance on your local host, which can be opened in the browser window. Where it says “Host”, enter the swarm manager IP with port 8000.

In the other swarm manager instance, run the autoscaler by running `python3 autoscaler.py`. The program will prompt you to enter the swarm manager IP. The program will then ask you if you wish to enable or disable auto-scaling. Once you have made your selection, click “Start Swarm” on your Locust browser. As Locust simulates client users over time, you will start to see results in the ‘Charts’ tab with information such as “Total Requests per Second,” “Response Times,” and “Number of Users.” As the autoscaler runs, a file called “application_size_plot.png” will be created in the project folder, which is a plot of the number of containers over time.

Once the run stops, you may start a second run by clicking the “New” button in the top right of the Locust page.

Conclusion

In conclusion, the implementation of the auto-scaler for cloud microservices demonstrates the feasibility of dynamically adjusting application scalability based on workloads. By monitoring the response times the autoscaler is able to employ its scaling decisions, whether to scale out or scale in, to maintain efficiency and reliability in modern cloud computing. Throughout this project, we familiarized ourselves with technologies that helped achieve the autoscaler’s objectives. Specifically the Docker Python SDK, Requests and Matplotlib libraries were utilized in our code, and GitHub was used within our team to organize ourselves to complete the tasks. Gaining knowledge in an auto scaler’s design principles and utilizing the technologies, we were able to successfully implement an autoscaler microservice of our own, that consists of visualization functionality (real time plots).

References:

Locust:

https://docs.locust.io/en/stable/index.html

https://github.com/locustio/locust/tree/master/examples/custom_shape

Docker Python SDK:

https://docker-py.readthedocs.io/en/stable/index.html

PyPlot:

https://matplotlib.org/stable/tutorials/pyplot.html

Requests:

https://pypi.org/project/requests/

Time/Datetime:

https://docs.python.org/3/library/time.html

https://docs.python.org/3/library/datetime.html