ECE 422 Winter 2021

Project 2: Reliability
Auto-scaling for Cloud Microservices

Andrew Williams & Jordan Los

April 18, 2021

Abstract

Cloud computing revolves around the ability for cloud services to efficiently compute a large number of external requests. One of the ways in which this efficiency can be found is by horizontally scaling the allocated resources in or out to adjust for the load on the server. In this lab we developed an algorithm which automatically scales the number of replicas for a web service in docker swarm in order to manage the response time of the server. This was done through an auto scaling algorithm designed to estimate the number of replicas needed by the swarm and adjust as required. History plots representing the performance of the auto scaling ability are then displayed in order to visualize the success of the auto scaler.

Introduction

The modern transition towards cloud computing has revolutionized the way we use computers. Cloud computing has caused the accessibility of storage, networking, and processing power for the average person to skyrocket. It has also enabled businesses of all sizes to avoid cost and logistics of managing their own IT infrastructure. One of the key benefits of cloud computing is the optimization that can be achieved by running many tasks on one machine, such as less downtime wasted for hardware and more efficient power usage. For this to work as efficiently as possible it is important that the services provided by the cloud infrastructure can be scaled up or down to meet the current needs from the clients.

In this lab, we provide an automated solution to the issue of variable cloud demand. For the purpose of this example, Docker swarm is used as the manager for the cloud microservices. To enable auto scaling we deploy a service that runs in the swarm manager that automatically scales the amount of services run by the workers based on the response times of the services. The estimated number of required replicas is calculated and then the swarm is scaled appropriately. The data collected by the auto scaler is then displayed in real time to a webpage in order to show the effect of the auto scaling service.

Technologies, Methods, & Tools

Python

Python was chosen as the language to automatically scale microservices to the demand of the clients. This provided us with a simple method of implementing the auto scaling algorithm, as well as enabling us to feed this information to a webpage for visualizing the data.

Docker and Docker Swarm

Docker containers provide a great solution for servers because they offer a lightweight, standalone, executable package that includes everything needed to run an application. Docker also provides the ability to group machines together. This allows one machine to manage the rest and enables efficient collaboration. We used docker swarm as a method of implementing our microservices and used the build in scaling commands in our auto scaling algorithm.

Cybera

In order to host the Docker hosts used in the swarm, we used Cybera cloud virtual machines. We used three VMs in total to demonstrate the auto-scaling. One VM was the swarm leader, one was a worker node, and the last was the client. Cybera allowed us to test the auto scaling in an environment that more accurately represents the way a cloud would be deployed. Because of the flexibility of Docker, the cybera infrastructure is not required to run this application, but is needed to allow the Dockers swarm to scale up as intended.

Redis

One of the microservices being run by the Docker swarm is a datastore service using redis. Redis allows for quick in-memory data storage. This microservice is responsible for keeping a log of the number of hits that the swarm cluster is getting. The Redis datastore is then used to retrieve information for the auto-scaling algorithm and the real-time presentation of data.

Flask, WebSocket, JavaScript, and Plotly

A real-time web application is one of the services being provided by the Docker swarm. This web application visualizes the effects of the auto scaling, and is a way to monitor the workload of the swarm. This web application is partially run through a python script that runs alongside the auto scaling algorithm using Flask. It is also partially run in a JS script within the HTML of the page. WebSocket is used to transfer the necessary data between these scripts.

Design Artifacts

The following design artifacts can be found in the git repository under the folder “Design”. The contents of each of the three files in this folder are explained further in this section.

Architectural View

The architectural view file shows a high level view of the interaction between the elements in order for the auto scaler to work. In this diagram we can see the four white boxes as the service types available for the docker swarm. On the left is the auto scaler service where the scaling algorithm and performance plotting takes place. To the right of the auto scaler is the visualizer provided by docker to view the active services and their hosts. To the right of the visualizer are the web services. The web services are the actual work being performed by the client, and as such there is a request and response relationship between them and the client. The number of these are what will be scaled up and down by the auto scaler. Lastly on the right is the redis service that records the amount of hits to the docker swarm services. The interactions between these services are shown by the arrows. The node boxes beneath in two shades of grey represent which services will run on which nodes. It should be noted that the Manager node is also a worker node and so can run them all.

State Diagram

The state diagram file shows the process that the auto scaling algorithm goes through to scale the docker swarm services. In this file, the small circle on the left is the starting point. Before the autoscaling loop, there are a few things that must be initialized. After this, the metrics are pulled which will be used by both the auto scaler and the visualization plots. After these metrics are retrieved and calculated they are compared with the pre-defined threshold for acceptable response times, and the auto scaler will either calculate a new number of required replicas or not. Either way, the visualization plots are updated with the new metrics, and the loop restarts.

Pseudocode

The auto scaler algorithm is described in the pseudocode text file. This algorithm first takes metrics required such as the workload and the response time of the docker swarm services. Then, the function checks whether the response time is outside of an acceptable range. If it is, the auto scaler will compute the estimated number of replicas needed by the docker swarm in order to manage the workload. This is found by targeting a certain number of seconds per request, and with this and the frequency of hits, calculating the number of required replicas. This number is then used to scale the swarm. After scaling, the graphs are then redrawn in order to show a real time visualization of the swarm scaling.

Deployment Instructions

On the cybera cloud download the updated docker-compose.yml using
`wget`
`https://raw.githubusercontent.com/JordanLos/ECE422-project2/master/docker-compose.yml`

Then run the command sudo docker stack deploy --compose-file docker-compose.yml app

All the necessary files are in docker images on DockerHub, so the compose file will download and run everything as necessary.

Conclusion

The efficiency of cloud services is in the interest of almost anyone in contact with a computer. As such, there is constant research and development going on to find new methods and technologies to improve them. One simple, yet fundamental technique is scaling the amount of resources used by a cloud server to match the requirements being asked of it in real time. In this lab, we showed one example of how this could be done through an automated Docker service that scales the amount of services being used in the Docker swarm based on a workload and response time algorithm. In doing this, we demonstrated the effectiveness of auto scaling, and visualized the results in real-time.

References

The ECE422-Proj2-StartKit was used as the base of our code, provided by Hamzeh Khazaei on github: https://github.com/hamzehkhazaei/ECE422-Proj2-StartKit

The following articles and websites were used for various technical and architectural guidance:

- https://docs.docker.com/engine/swarm/
- https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/
- https://docker-py.readthedocs.io/en/stable/client.html
- https://github.com/dannyvai/plotly_websocket_example