ECE422 Auto-scaling for Cloud Microservices Project

April 17, 2021

Hao Xuan 1537913

Maocheng Shi 1538829

Abstract

This project is to implement a reactive auto-scaling engine for a cloud microservice application. This application is implemented using Python and deployed on the Cybera infrastructure using Docker microservices. The auto-scaler developed in our project can scale out or scale in the application according to the workload. The autoscaling application monitors the client requests and automatically adjusts capacity and number of processes to maintain steady, predictable performance at the lowest possible operational cost. This application includes web service and datastore service, in this project we mainly worked on the web service to improve its performance. We also developed our algorithm to avoid ping-pong effect that may occur in auto-scaling applications.

Introduction

In many situations, the workload of a server which provides the webservices is not the same for all the time. In some situation the server receives a few requests from the clients and only need a little computation resource. However, sometime the server that is rarely accessed may suddenly receive a huge volume of workload concurrently. Beartrack system that is used for enrolling courses can be an example, students do not visit it unless the courses start enrolling but when the enrolling date the huge amount the of requests exceeds the capacity of the server and the respond time become much longer than usual. It is difficult to deal with such situations if we allocate a fixed number of computational resources for each webserver. If we try to allocate enough resource for each server to satisfy the worst situation, a huge amount of resource is wasted and the webservice provider shall need much more money to buy and maintain the servers. If the computational resources allocated for each webserver is not enough, in the worst situations the server respond time become very long and may stop the service.

In this project we implemented a reactive auto-scaling engine for a cloud microservice application to power the webservers. We aim to optimize the performance and the cost of the webserver at the same time and try to make the webserver has a fixed amount of respond time regardless of how many client requests it receive.

Technical Details and Methodologies

In this project we implemented an auto scaling application and a client application for testing. The most challenging part of this project is to implement the auto scaling program for the server. It works by monitoring the total number of client request and count how many of the client requests exceeds the limits. To avoid oscillation (ping-pong effect), the auto-scaler counts the request it received and update the server scale every 5 requests. If 3 out of 5 requests takes longer than the upper bound of the time

limit, the scaler automatically increase the server scale by 1. Similarly, if 3 out of 5 requests takes shorter time to respond than the lower bound the auto scaler will shrink the service size by 1. In our design we used one function to monitor the requests and another function to automatically scale the service.

```
33      def monitor(self):
34          count = 0
35          s = 0
36          while(1):
37              count += 1
38              try:
39                  r = requests.post('http://10.2.10.38:8000/')
40              except:
41                  time.sleep(1)
42                  r = requests.post('http://10.2.10.38:8000/')
43
44              s += r.elapsed.total_seconds()
45              if(count == 5):
46                  print(f'{s/5}')
47                  redis.rpush('workload',s/5)
48                  self.auto_scale(s/5)
49                  s = 0
50                  count = 0
51              time.sleep(1)
```

Figure 1 The function monitors the requests

.

```
53      def auto_scale(self, t):
54          s = self.check_threshold(t)
55          if (s == 1):
56              self.size += 1
57              self.web_service.reload()
58              self.web_service.scale(self.size)
59              redis.rpush('scale',self.size)
60              time.sleep(2)
61          elif (s == 2 and self.size > 1):
62              self.size -= 1
63              self.web_service.reload()
64              self.web_service.scale(self.size)
65              redis.rpush('scale',self.size)
66              time.sleep(2)
```

Figure 2 Auto scaler function

The server scale and the number of hits on the webpage shall be stored into the database.

We used flask-dashboard frame to build a webpage which shows the real time data. In the webpage it has three charts, one chart shows the number of requests per second, one chart shows the response time and the other one shows the server scale. The host server of the webpage retrieves the data from Redis database and reflect the values on the webpage.

Design Artifacts and Explanation

In the GitHub repository it contains a file named “auto_control.py” which is used to monitor the request that the server received and automatically scale out or scale in the application. The folder named flask-dashboard contains the server for holding the webpage which shows the real time data. The file named “http_client.py” is used to simulate the behaviour of many users and add workload to the server.

In the design folder, it contains a high-level architectural diagram of this project and the state transition diagram for explaining the mechanism of the auto scaler. In the architectural diagram, it shows a http client, the auto scaler a web application and the Redis database. The auto scaler monitors the requests sent from the http clients and control the application to scale in or scale out according to its response time. The application store and retrieve the data from the Redis database. In the state diagram, it shows the auto scaler has three states. One state is monitor state, the auto scaler monitors the number of requests it received and count the requests from 0 to 5. If more than 3 out of 5 of the requests has longer response time than the upper bound it goes to the scale out state to give the server more computational resource. After scaling out the application the monitor reset the counter and count them from 0 to 5 again. If 3 out of 5 requests ha less respond time than the lower bound, the auto scaler scale in and release some computational resource.

Deployment Instruction

First, follow the instruction given in https://github.com/hamzawey/ECE422-Proj2-StartKit to set up the swarm cluster and the base application.

Then use the following instruction to deploy the Auto Scale service.

```
$ git clone https://github.com/lovettxh/Cloud-Auto-scaling.git
```

To start the monitor service:

```
$ cd flask-datta-able
```

Then install all the requirement by using command

```
$ pip3 install -r requirements.txt
```

Then set the flask running environment by using the command

```
export FLASK_APP=run.py
```

Finally start the server by

```
$ flask run --host=0.0.0.0 --port=5001
```

To deploy the auto scaling service:

```
$ pip3 install redis
$ pip3 install requests
$ pip3 install docker
Python version needs to be at least 3.6
```

Then run the service by:

```
$ sudo python auto_control.py
```

Conclusion

In this project we developed an auto scaling application which can scale in or scale out according to its workload to optimize its performance and cost. The auto scaler meets the requirement specified in the project description and we get the knowledge on developing self-adaptive application. In this project we also practiced using Redis database and flask framework to implement high performance web server in high concurrency.

Reference

Wikipedia contributors. (2021, April 4). Redis. In *Wikipedia, The Free Encyclopedia*. Retrieved 00:05, April 19, 2021, from https://en.wikipedia.org/w/index.php?title=Redis&oldid=1015986749

Wikipedia contributors. (2021, April 12). Docker (software). In *Wikipedia, The Free Encyclopedia*. Retrieved 00:06, April 19, 2021, from https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1017343533

Github contributors. (2020, June 1). Flask-data-able, retrieved from https://github.com/app-generator/flask-datta-able.git