Auto-Scaling for Cloud Microservices Project

ECE 422 Lec B1 - Winter 2023

Reliable Secure Systems Design

Members: Owen Stadlwieser and Martin Rudolf

Abstract

As applications scale, they need to remain feasible for use by both client side-users and server administrators. Part of these conditions include needing to maintain reasonable response times so that users can expect fast service for their requests and so that their requests are reliably serviced. For server administrators, it is important to remain cognizant of the potential resource constraints associated with having dedicated compute capacity to service these requests. Particularly, when the number of user requests is low (for example at night), it's important that resources are freed up and not sitting idly by and consuming electricity and ultimately money.

Introduction

In this project, we implement a reactive auto-scaling engine for a cloud microservice application that balances the needs for reliability and performance. Using Docker containerization, we implement a program that reads the number of incoming requests and scales the available resources *in* or *out* depending on whether the response times fall below a lower threshold or exceed an upper threshold, respectively. This method of horizontal scaling allows us to scale *in* and keep the operational cost of running the application low or scale *out* when needing to maintain the reliability and performance. These approaches are balanced and implemented in our project in a self-adaptive manner such that the microservice should be able to respond and handle a wide-variety of requests and workload shapes.

As a starting point for our implementation, we inherited the ECE422-Proj2-StartKit and built out our application based on this skeleton code. In this basic code, the application is seen to consist of web and datastore microservices. Among these two, our solution focuses on the web microservices as it has the performance bottleneck.

Methodologies and Technologies

Methodologies

Our solution implementation started with us researching how to apply horizontal scaling in Docker. We implemented a docker auto scaling service which runs as an application gateway, and forwards requests to

docker swarm workers. Once a user requests a service, the request hits our gateway at which point it immediately passes the request to a worker. The worker can then independently of the gateway perform the computational effort for the user's request. This leaves the gateway free to handle thousands of requests as the only computation effort occurring within its API route consists of appending the response time of the worker to an array. We decided on this solution because it decouples the autoscaler and the worker API endpoints into separate modules. Furthermore, this solution gives horizontal scalability, and does not require altering the API, or client modules, quietly integrating the gateway into the application without disrupting the current architecture.

Alongside the API, our gateway has two worker background threads which control both the drawing of a *Number of Workers vs Time* graph, as well as controlling the number of worker replicas. The latter task being the most important. The check loop computes the average response time over the last 10 seconds. If the average response time exceeds 2.05 seconds the check loop thread increases the number of worker replicas. On the other hand, if the average response time is less than 1.585 seconds, the number of workers is decreased.

To create more effective decrementing, we decrement one replica at a time. We believe this to be an appropriate use of applying “memory” to the decrement decision so that after a spike in demand, the replica count slowly decreases and remains available if another spike occurs shortly after the preceding one. To note, the number of replicas will never decrease below one so that some level of service will always be maintained so that when any client makes a request the system will be ready to service it. For incrementing, the auto scaler calculates the average response times in the window divided by the baseline average and increases the number of replicas by that amount. This allows for more reactive scaling out and quicker performance and reliability gains.

We performed an experiment to calculate the expected lower bound on response times for our load balancer. Using one replica, we sent 100 requests from the load balancer to the single replica. From these 100 requests we determined the expected response time for a system with no backlog. This average serves as the lower bound for the response time. From this minimum any response times within a small range (gamma epsilon) indicate the amount of replicas in the current system is excessive. We also calculated the standard deviation which will be further expanded on later.

After performing this experiment, we considered whether our decrementing should depend on requests/sec instead of response time. Since responses were handled by one worker at a time, there came a point where no greater gains could be made if each request was being handled by a worker and requests/sec was giving a more meaningful representation for our application. We chose against requests/sec though as it wouldn’t be a representative metric when the response time varies depending on the nature of the request, as it would in a real world situation.

We also worked to avoid the ping-pong effect which would reduce our solution effectiveness as it would quickly “ping-pong” between different replica counts by incrementing and decrementing successively. Part of our solution was to compare the response times from the previous 15 seconds in our rolling window against our baseline response time average (which we defined to be 1.7 seconds). If the previous window average exceeded the baseline average by some factor ($\epsilon = 0.35$ seconds), then we would

increment the replica count. If the previous window average fell below the baseline average by some factor ($\gamma = 0.11$ seconds), then we would decrement the replica count. These upper and lower thresholds were determined through the above outlined experiment and through informal trial and error as we were discovering this problem domain.

One particular priority was reliability, so if a large number of users join at once, our replica count might overshoot and produce more than are needed. After the request backlog is clear, it will stabilise by removing the unnecessary replicas one at a time. Decrementing also operates on a sustained decrease in load so that it decrements then checks to see if the new lower replica count is still sufficient and can be lowered further, this creates a step down effect.

One shortcoming of our implementation is that the service starts at 1 replica and only scales after the first window (in our case 10 seconds) has finished. This means that there is the potential for failed requests or slow response times during these first seconds. Afterwards, the requests are expected to perform reliably with an error rate of no more than a few percent.

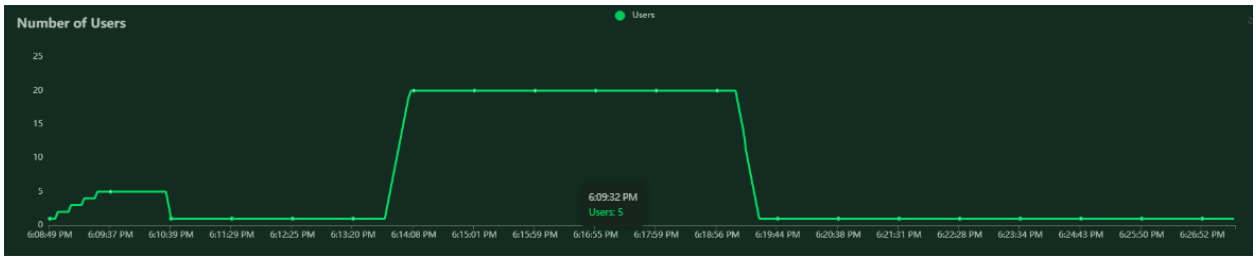
Technologies

- Docker: used for auto scaling replicas of our Flask app.
- Python: our solution was coded with Python3 using additional libraries outlined in our requirements.txt file. Some of the major packages used include:
    - Flask: a web microframework for handling server requests from clients.
    - Docker SDK: for interacting with our docker client.
- Locust: for workload testing and visualisation. We use this to simulate a bell-shaped workload on our autoscaler and here we also see requests/sec, response times and number of users being simulated.

Results

After running Locust to test our microservice with a bell-shaped workload curve ranging from 1-20 users, we found that the response times consistently settled at an average of approximately 1.7 seconds. Our algorithm worked effectively and scaled as expected. Below are some screenshots of a simulated workload:

Figure 1: Number of Users

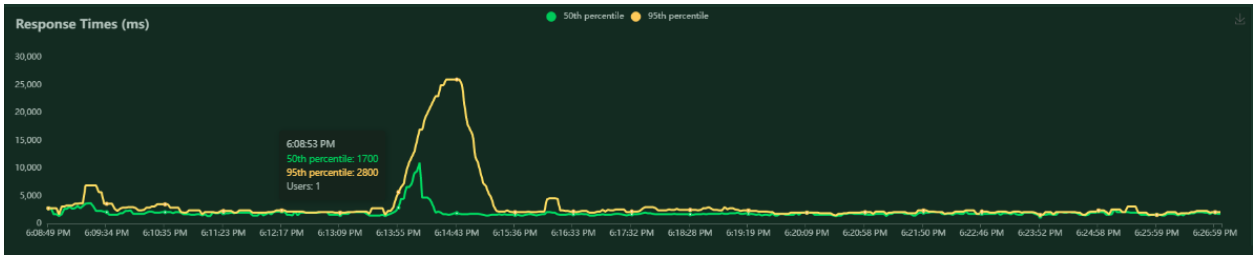


In this simulation the number of users went from 1 to 5 to 1 to 20, and finally back to 1.

Figure 2: Requests per second

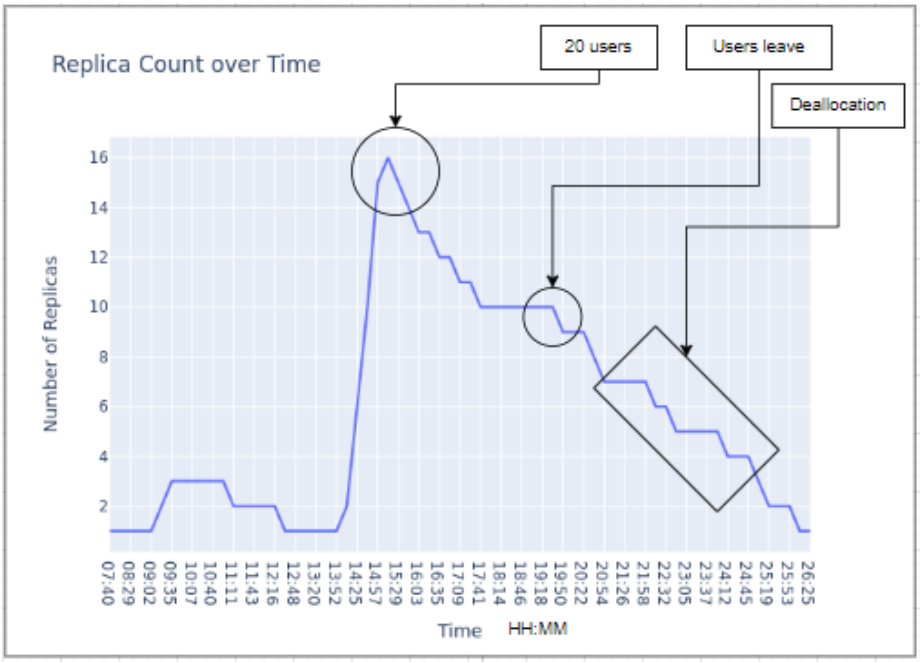
The requests per second is proportional to the number of active users at one time. The green line being requests per second, and the red line represents the number of failed requests.

Figure 3: Response times.



The gold line represents the 95th percentile of response times, the green line represents the 50th percentile of response times. As can be seen, there was a clear backlog of requests when the 5, and 20 users became active. The load balancer was able to stabilise the response times while the users were still active. Furthermore, the load balancer kept the system stable after users began leaving the service.

Figure 4: Replica count over time

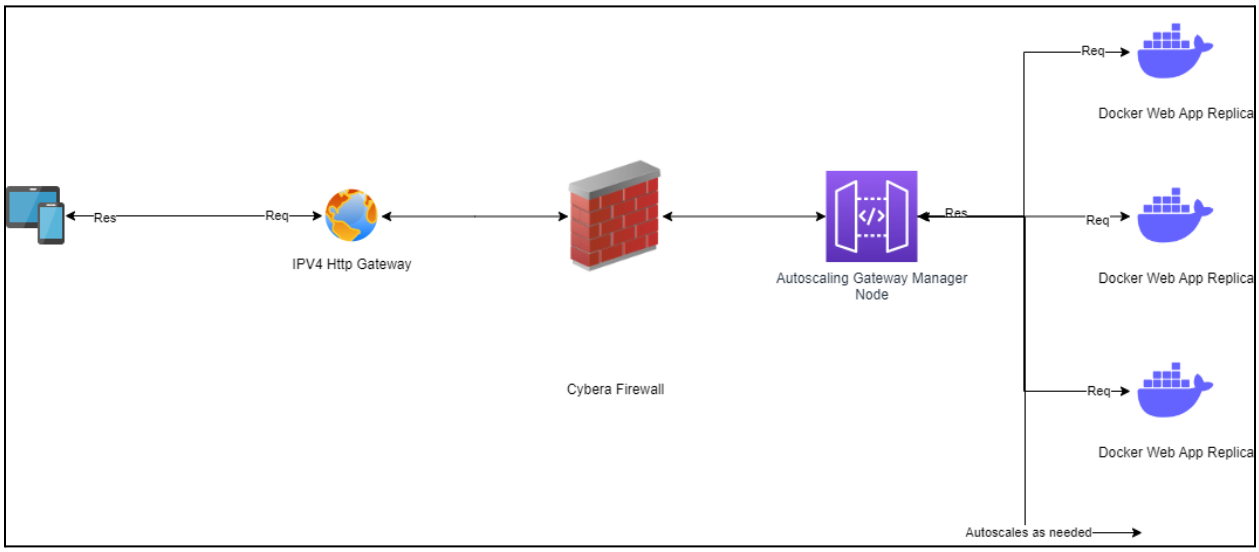


The replica count over time can be seen to increase during the times at which many users became active. After this time the server found a period of stability, or took replicas offline in the case that there was an overshoot. In the case of 20 users joining all at once, 16 replicas were allocated, replicas were taken offline until a period of stability at 10 replicas. To prevent request failure we allocate based on the amount of users that join at once, instead of allocating one replica per window. Finally, after the users became inactive, the server began deallocating replicas, one at a time. We utilised sustained decrease in load over time instead of deallocating all ressources at once, this design handles better real world scenarios where users will frequently leave, and rejoin a service, like the intermittent use around a real world event like the World Cup on Twitter. Also by setting our lower band for deallocating resources within one standard deviation of the expected mean response time, the variance in response times allows for replicas to be deallocated. Response time varies over windows, and will be below the mean within one or two windows when we are overallocated, causing a step down effect, and keeping the system more stable.

Design Artefacts

High-level Architectural View

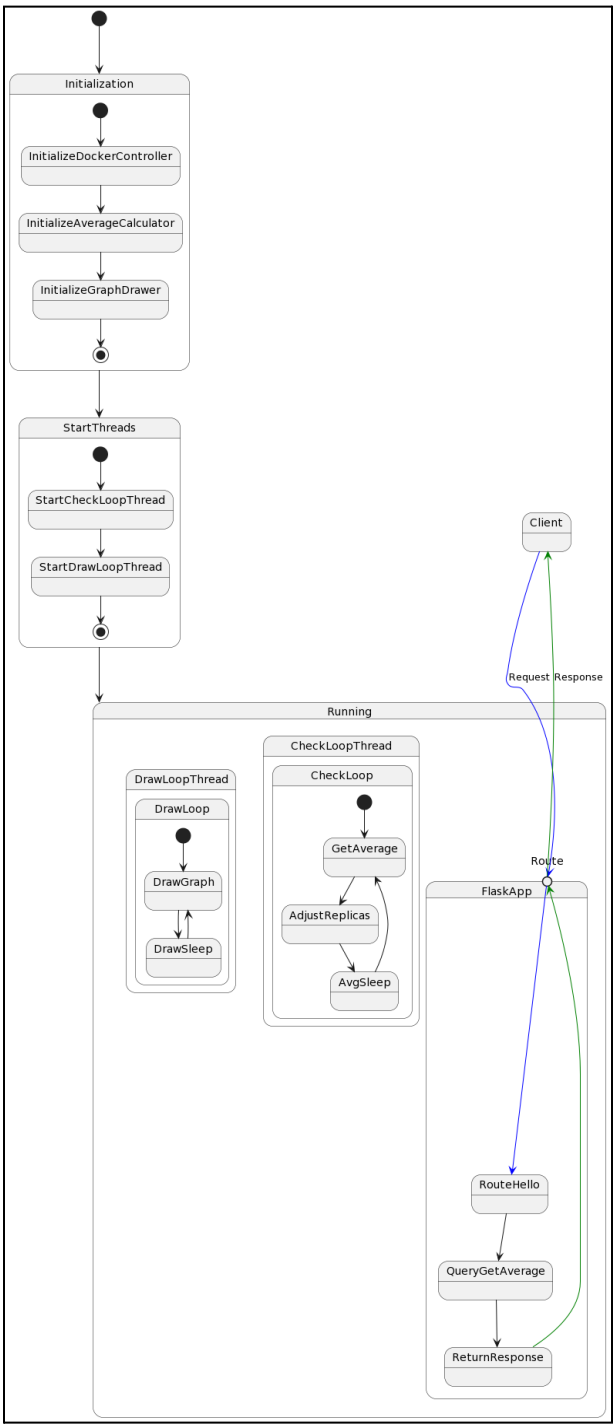
Figure 6: Architectural view of our cloud microservice



From the left, we see a group of devices make a request through a gateway which itself passes through the Cybera firewall. Once it hits our autoscaling gateway, it will pass on those requests to a replica of the web app. That replica will service the request and pass it back to the autoscaling gateway which will then pass the response down the same pathway as the request came, just in the opposite direction.

State Machine

Figure 7: State Machine for the autoscaler cloud microservice



Shown here is the State Diagram for our autoscaler. The program begins with initialization of the main classes and starting threads for updating the number of replicas (CheckLoop) and drawing a real-time graph for the number of replicas (DrawLoop). Then the system moves to the running state. Here the two

threads run their functions periodically and clients can make requests which are handled by the Flask app and returned as responses back to the client.

Autoscaling Algorithm

Base_line = 1.7
Gamma_epsilon = 0.115
Delta_epsilon = 0.35
Monitor_interval = 15 seconds
While true:
    If response_times exists in window
        Average = average of response_times
        Clear response_times
        If average < base_line - gamma_epsilon:
            decrement_num_replicas()
        Else if (average > base_line + delta_epsilon)
            Factor = average // base_line
            increment_num_replicas_by_factor(Factor)
        Previous_average = average
        Wait for monitor_interval

Deployment Instructions

Prerequisites:

1. Create 3 VMs on Cybera cloud with the following specifications:
    a. Use `Ubuntu 18.04` or `Ubuntu 20.04` as the image for all VMs.
    b. You need one of these VMs to run the client program for which you may use `m1.small` flavor. Let's call this VM as the `Client_VM`.
    c. For the other two VMs, please still consider `m1.small` flavor. These two VMs will construct your Swarm cluster.
    d. You need to open the following TCP ports in the `default security group` in Cybera:
        i. 22 (ssh), 2376 and 2377 (Swarm), 5000 (Visualization), 8000 (webapp), 6379 (Redis)
        ii. You can do this on Cybera by going to `Network` menu and `Security Groups`. (See Here)
2. On the `Client_VM` run
   `$ sudo apt -y install python-pip`
3. `$ pip install requests`

4. Then, you need to install *Docker* on VMs that constitute your Swarm Cluster. Run the following on each node.
   `$ sudo apt update`
5. `$ sudo apt -y install docker.io`
6. Now that Docker is installed on the two VMs, you will create the Swarm cluster.
   a. For the VM that you want to be your Swarm Manger run:
7. `$ sudo docker swarm init`
   a. The above `init` command will produce something like the bellow command that you need to run on all worker nodes.
8. `$ docker swarm join \`
9. `--token xxxxxxxxxxxxxxxxx \`
10. `swarm_manager_ip:2377`
    a. Above command attaches your worker to the Swarm cluster.

Source: https://github.com/zhijiewang22/ECE422-Proj2-StartKit

Steps:

Note the first 7 steps can be performed by bash.sh in autoscaler directory

1. Download the code submission on to your swarm manager
2. `Cd autoscaler`
3. `sudo docker stack rm app_name`
4. `sleep 5`
5. `sudo docker build -t auto:1 .`
6. `cd ../`
7. `sudo docker stack deploy --compose-file docker-compose.yml app_name`

User Guide

Prerequisites:

1. Wait for deployment to complete

To send request to workers:

1. Send requests to “http://” + SWARM_MASTER_IP + “:8000/”

To generate number of workers plot:

1. Get id of container using `docker container ls`
2. `docker cp CONTAINER_ID:/code/replicas.png ./code.png`

Conclusion

For this project, we were able to implement a feasible solution for reactive horizontal auto-scaling of a web microservice. Using a swarm manager to scale the number of replicas of our service, we were able to decrease long response times as needed to improve performance while also being able to scale down to lower operational costs when fewer requests were being made to the webservice. We found a rolling window average over the response times to be a valid and effective solution to monitoring response times while reducing the likelihood of the ping-pong effect. While further scaling and more formal verification

could be beneficial in a commercial setting, we have found our solution to be effective for the suggested test cases and requirements of this project.

References

- https://docker-py.readthedocs.io/en/stable/
- https://docs.docker.com/