Reliability Auto-Scaling Report

Authors: Michael Boisvert, Wonbin Jeong

Abstract

For this project, we created a reliable system which has an implementation of an auto scaling system. The auto scaling system provides the server the option to scale its service up and down by monitoring the amount of requests and work done by the server. A reliable system should be able to meet users' demands and scale automatically on heavy workloads. We built such a reliable system using Docker Swarm, Python, and a Docker API using Python, called Docker-py.

Introduction

As systems are built on the cloud, it is essential to acknowledge that the server may be in a situation where it is faced with heavy workload. To make a reliable service, it is necessary to prevent server overloading and completely stopping a service due to the result of overloading (eg. Eclass). Such disasters can be prevented by the use of scaling systems. For this project, we will focus on horizontal scaling, which is a method to add or decrease the number of nodes required for a service depending on the workload the server is facing.

This project is hosted on Cybera, and utilizes the auto scaling solution we implemented to build a reliable system. Our service enables users to make a number of requests to our web service, using an HTTP request. If too many users are making too many requests at once, the auto scaling system should calculate the demand and act based on that, by scaling up (adding nodes) to the swarm. On the other hand, if there are a small number of users making too few requests, the server should scale down (decreasing number of nodes).

This report will be discussing the methodologies, design and implementation of how our team developed such methods for the project.

Technologies

- Docker & Docker Swarm
- Python
- Flask
- Plotly Javascript open source library
- WebSocket
- Redis
- HTML
- Docker-py
- Cybera

Methodologies

Our service is hosted on Cybera, using a Docker swarm cluster. A swarm cluster holds many lightweight virtual machines that are grouped together in a “cluster”. It is a great way to manage a service, as one machine can manage the group of virtual machines. We support 5 instances as follows:

- Redis:
    - This service is simply used for getting the number of ‘hits’, or the number of times clients have visited the web service.
- Visualizer:
    - This service enables users to visualize how many vm instances there are in each node, consisting of the manager node and worker node.
- Web:
    - This service hosts the client side, and lets clients send and receive requests from here.
- Auto scaler:
    - The auto scaler is the core of this system, which calculates the amount of workload the server is facing, and acts accordingly.
    - The auto scaler also sends the statistics of the server information via a web socket to the next service, the auto scaler plots.
- Auto scaler plots:
    - This service hosts a simple web application, and prints out plot information that is received by the auto scaler through a web socket.

Measuring the workload:
As mentioned, we use the auto scaler service to measure the amount of workload the server is facing. We do this by setting a monitoring interval, and observing the average response time during this interval. If this average response time is greater than the upper threshold, then the service is scaled upwards. On the other hand, ifi the average response time is less than the lower threshold, the service is scaled downwards. More details can be found in the pseudo code below in the design artifacts.

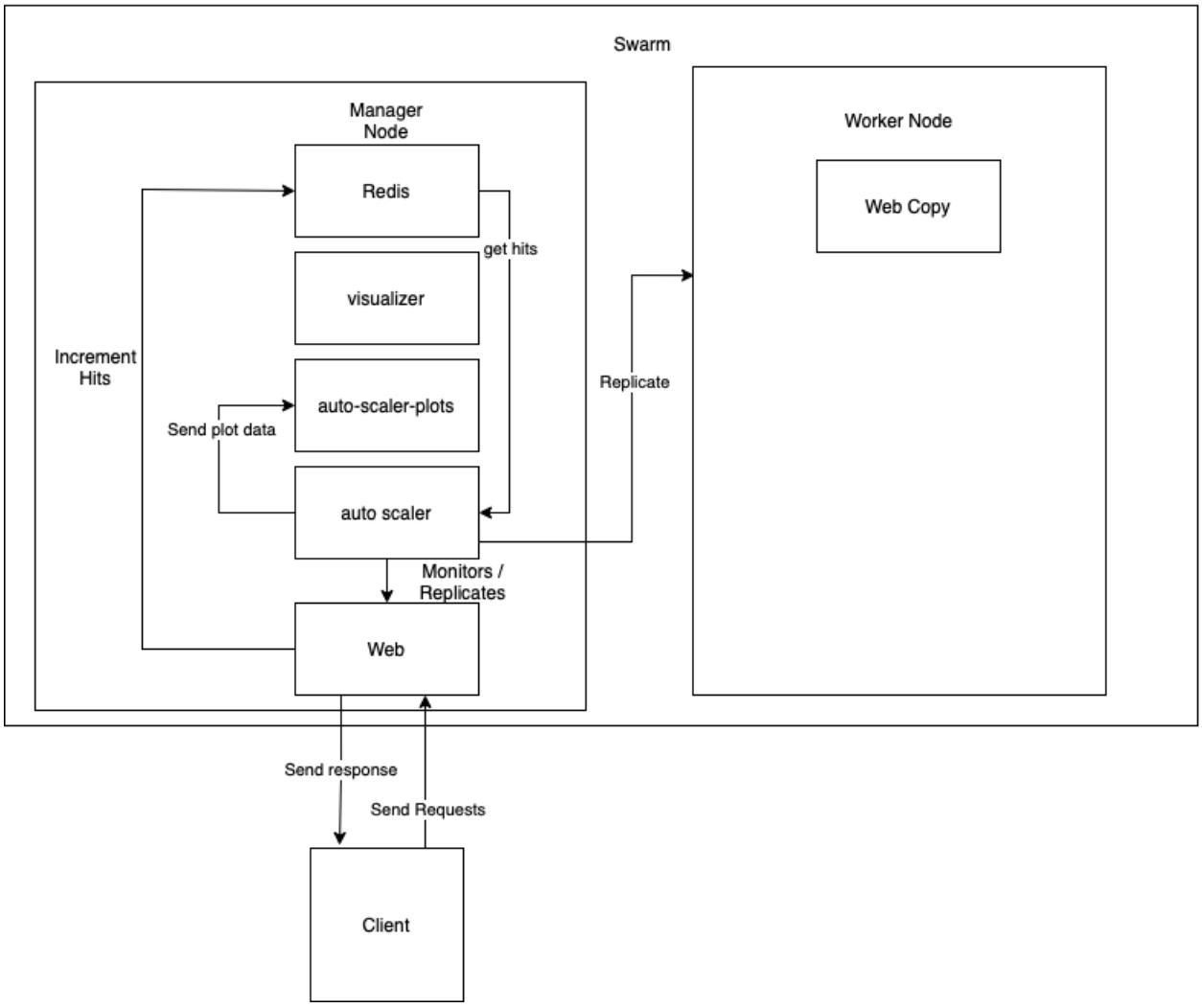
Scaling
Our service uses a package called Docker-py, which enables scripting of Docker scripts using Python. We made an independent auto scaling service, so that its computations do not affect the web service by slowing down the service.

The scaling can be toggled on and off using a button present in the plotting service.

Plotting
We used a library called Plotly to plot the workload, response time, and number of replicas of the web service. The data is received from the auto scaler, which is connected using a web socket instance.

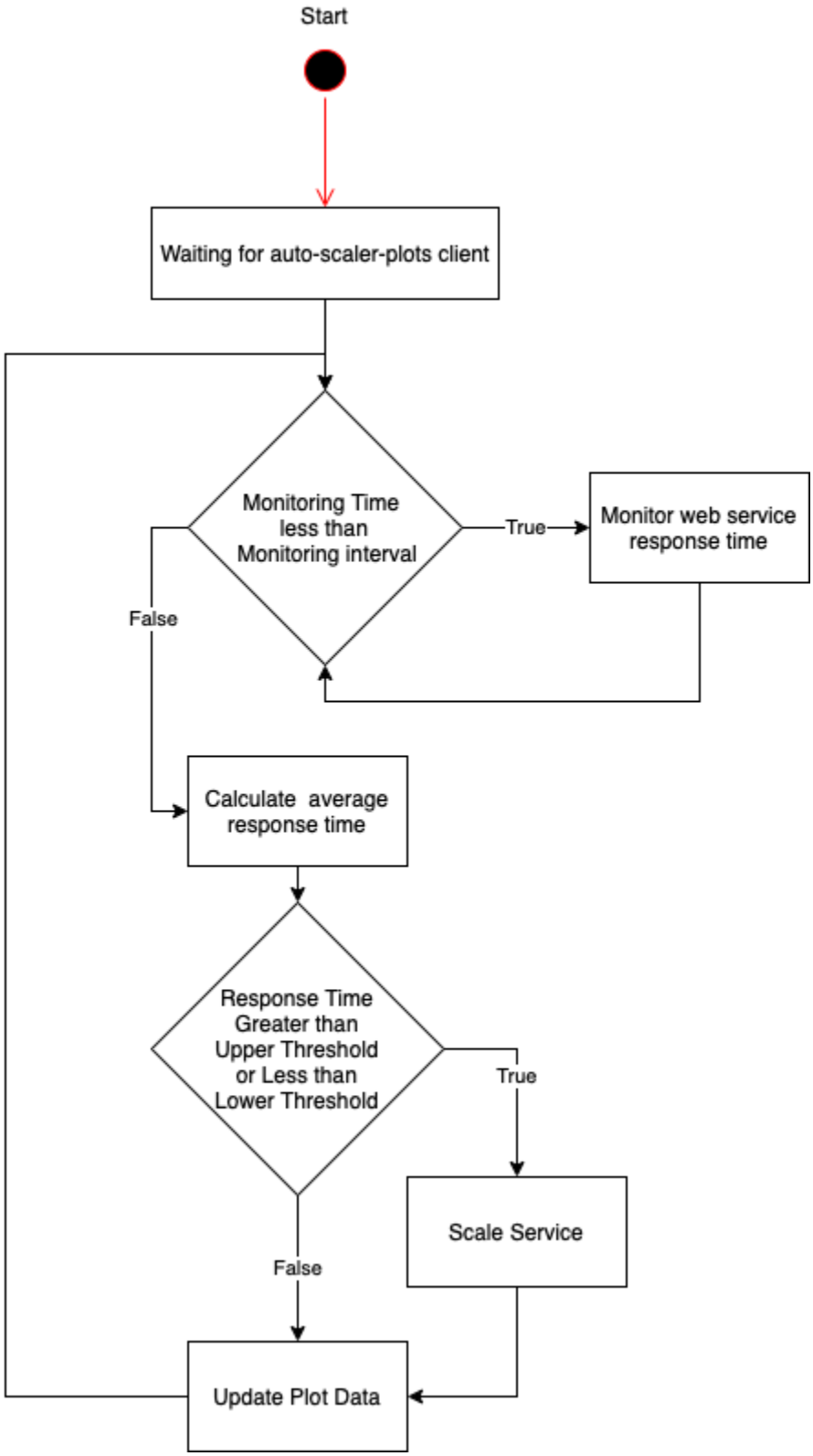
Design Artifacts

1. Architecture



As discussed in the methodology section, our architecture is composed of 5 instances, which are grouped together in a cluster. If the demands meet, the auto scaler scales up by creating replicas of the web application, and scales down otherwise.

2. State Diagram



This state diagram shows the logistics of how our auto scaler works.

3. Pseudocode

```
upper_threshold = X
lower_threshold = Y
monitoring_interval = Z
req_per_container = 3

function auto_scale() {
    while True:
        start_monitoring_time = get_current_time()
        response_times = []
        prev_hits = redis.get('hits')
        while(get_current_time() - start_monitoring_time < monitoring_interval)
            response_time = getRequestResponseTime()
            response_times.append(response_time)
        average_response_time = average(response_times)
        curr_hits = redis.get('hits')
        num_req = curr_hits - prev_hits
        if average_response_time > upper_threshold or average_response_time < lower_threshold:
            scale(num_req)
}

function scale(num_req) {
    num_containers = ceil(num_req / req_per_container)
    set_docker_replicas(num_containers)
}
```

As previously discussed, this pseudo code shows our original idea of how our auto scaler should work.

Upon testing, we noticed that a normal request time should take in between 2 - 4 seconds. Thus, we chose 2 as our lower threshold, and 4 as the upper threshold. We also noticed that performance starts to take a big hit when the web service has more than 7 requests, so we chose 7 requests as the value that a single container should ideally handle. As for monitoring, we decided to monitor the state every 10 seconds, hence the value of monitoring_interval being 10.

We monitor the application for 10 seconds, and gather the number of requests made, as well as the average response time of each request. If the average response time exceeds or is below the thresholds, we rescale the service to meet the demands.

Deployment Instructions & User Guides

Requirements:

- Python 3.8 installed
- Ubuntu 18.04 Linux machine installed
- Docker installed

Deployments:

- Build and deploy all of the Docker images using the bash script provided:
    - **Bash ./build_deploy.sh**

Conclusion

As cloud computing is becoming a predominant way of hosting and building services, it is extremely important to create stable and reliable services. One of the ways to build a reliable service is by using horizontal scaling methodologies. In this project, we took advantage of Docker Swarms and its ability to easily implement horizontal scaling by creating our own service, which handles automatic scaling. We were also able to visualize the effects of horizontal scaling, by plotting its results.

References

- Docker-py: https://docker-py.readthedocs.io/en/stable/
- Plotly: https://plotly.com/python/getting-started/

- Docker compose: https://docs.docker.com/compose/
- Web Socket: https://websockets.readthedocs.io/en/stable/