Secure File Systems (SFS)

ECE 422 LEC B1 - Winter 2023
Reliable Secure Systems Design

Auto-Scaling for Cloud Microservices

Software Project
By
Ashutosh Lamba
And
Kyle Bricker

Abstract

The focus of this report is on detailing and discussing the design process behind an autoscaler for a web application that is deployed through Docker. This autoscaler is reactive to the amount of incoming requests and will auto-scale itself to the desired load requirements.

Introduction

The goal of this project was to design, implement and deploy an auto-scaler for cloud microservices. This report will go over the methodologies and technologies used, as well as will cover design artifacts, and instructions to run and use the program.

The auto-scaler works by monitoring the response time of users' requests in order to determine whether the response times are within an acceptable range. If the times are too slow, the scaler scales out the application to maintain performance, and if the times are too fast, the scaler will scale in the application to reduce running costs. This gives a reliability that users can expect a certain response time from our service, no matter the number of users making requests to it.

Methodologies / Technologies

Client:

Our client utilizes locust to simulate a large number of users accessing the web application, as well as to record our response times and graph them. Our connection to the server is done via web sockets through pythons 'socket' module.

Locust

The locust module in python allows us to simulate multiple users accessing our web application, as well as recording and displaying data related to user HTTP queries.

Socket

Web sockets are utilized in order to communicate with the server.

Gevent

The gevent module is used to spawn events that we use to grab recent response times.

Server:

The server application can be broken down into two microservices:

- The webservice ‘myapp.py’ is a web application that performs difficult mathematics upon user requests.
- The datastore that keeps track of the number of tasks performed.

The bottleneck here is the webservice, therefore we need an autoscaler to account for the performance issues of this microservice.

Our autoscaler has a response time window of between 3000-5000ms, below this, we scale down and above this we scale up. In order to communicate with clients we utilize web sockets, and in order to access and scale our web microservices we utilize Docker.

Socket

Web sockets are utilized in order to communicate with the server.

Docker

The Docker module allows access to our Docker microservices, allowing our swarm manager to change the number of active microservices.

Matplotlib - pyplot

Matplotlib is utilized to create additional graphs from data collected by our autoscaler.

Design Artifacts

High-Level Architectural View

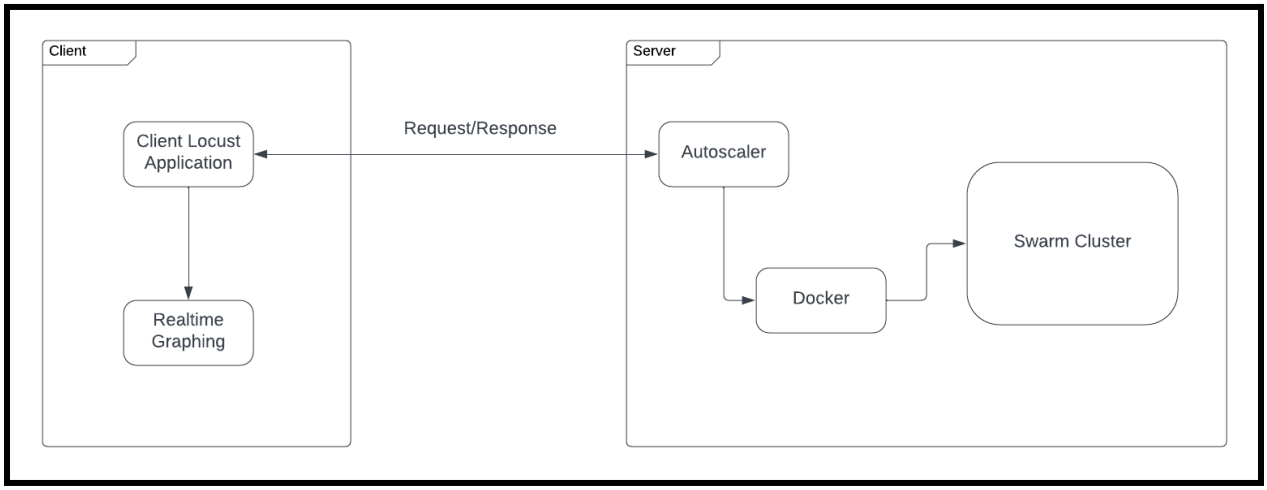

Figure 1: High-Level Architectural View

The high-level architectural view of our application can be seen in Figure 1. In this we can see that we have a server and client application, where the client makes requests to the server. The server then utilizes its autoscaler to maintain the performance of requests via Docker microservices and a swarm cluster.

Autoscaler State Diagram

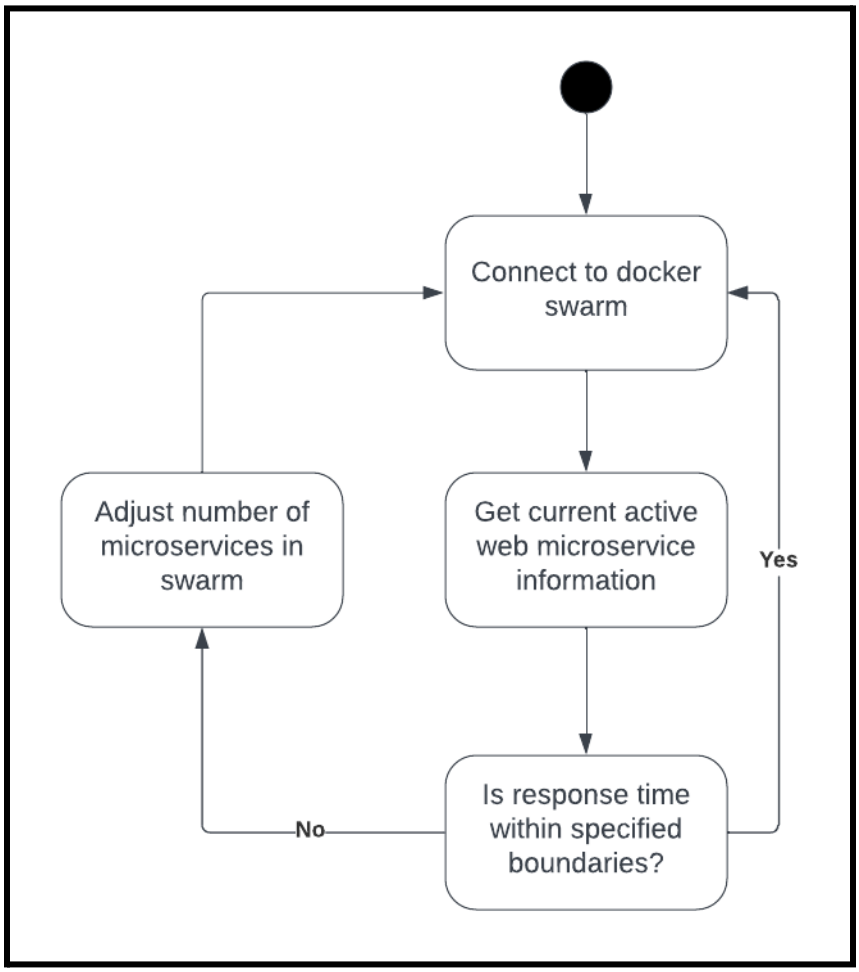

Figure 2: Autoscaler State Diagram

In Figure 2 we see a simplified representation of the states of our autoscaler. We see here that the autoscaler connects to the docker swarm, then on success we get and measure the response time of our web microservices, scaling our application based on our response time.

Autoscaler Pseudocode

```
"""
Pseudocode

get 99th percentile average response time
if average time > max_response_time
    scale up by ceil(response_percentile - max_response_time)/ 2000 (n processes for n users)
elif average time < min_response_time
    scale down by floor(response_percentile - min_response_time)/ 2000 (n processes for n users)
    as long as scale>0 (cant have 0 services)

Parameters

max_response_time = 5000.0
min_response_time = 3000.0

Explanation
in an ideal schenario (1 user per user) the response time is around 2000-2100ms max, thus in
an optimal schenario, the response time would be less than 3000ms and we would be safe to scale
down.
we went for a maximum response time of 5000ms as that means that the app is pretty overloaded
with more than 3 users per worker.
we also constrain that the minimum number of replicas must be 1 as we must have at least one
replica of the service
"""
```

Figure 3: Autoscaler Pseudocode

In Figure 3, we see the Pseudocode for our autoscaler. From this we see that our autoscaler takes a 99th percentile average of the response time, then it checks if this average is above or below our response time range. If it is below, we scale our application down to reduce cost, if it is above, we scale our application up to improve speed.

Deployment Instructions / User Guide

Deployment instructions for this application can be found in the README.md file, which is updated with instructions for our unique application.

Once the locust application and the server/autoscaler are both running, you can go to the specified IP (‘http://10.2.5.226:8089/’ for our machine) for the client locust application where the user can view the statistical graphs in real time. Users may also change parameters such as the number of users from this page.

Conclusion

This report outlines the process of designing, implementing and deploying an autoscaler for cloud microservices. The autoscaler monitors the response times of requests by users in order to determine whether it should scale up to maintain performance or down to reduce costs. This ensures that users will have a consistent level of performance when interfacing with our web application, regardless of the number of requests being made, which is essential to user experience. Overall, auto scaling techniques have proven to be a valuable tool for maintaining user performance while also minimizing cost.

References

- https://docs.python.org/3/library/socket.html
- https://docker-py.readthedocs.io/en/stable/
- https://www.gevent.org/
- https://docs.locust.io/en/stable/