Auto-Scaling for Cloud Microservices

By:
Lucas Zeng (zichang, 1539168)
Khang Vuong (kdvuong, 1532165)
ECE 422 LEC 502 - Winter 2022 Reliable Secure Systems Design

Abstract

In this project, we implemented auto-scaling for web services using Docker Swarm, Docker API, and Nginx. This report entails the implementation details, design rationale, and guides on setup and usage. The application is deployed using Docker Swarm which manages a cluster of Docker hosts. The auto-scaling capability is implemented through a custom python script that utilizes Docker API to scale the application up or down depending on the log time and response time of each incoming request. The log data is recorded by Nginx.

Introduction

The purpose of this project is to design and implement an autoscaler service that monitors docker service status and scale services up or down in an automatic manner. By implementing the autoscaler program to interact with Docker, we researched and practiced the foundational technologies of cloud infrastructure and gained a basic understanding of modern large-scale cloud services.

Technologies and Methodologies

Technologies

- Docker, an OS-level virtualization tool for service deployments
- Docker Swarm [1], a clustering platform based on Docker
- Nginx, a high-performance HTTP/Socket reverse proxy program
- Python programming language
- Docker-compose: a Python program that utilizes docker and provides systematic deployment configuration using yaml file
- Flask[2], a simple, performant Python HTTP library
- Docker Remote API for Python, used for interaction between python program and Docker cluster.
- Chart.js[4], a chart-drawing library for client JavaScript.
- Socket.IO[3], an abstraction library on top of WebSocket and HTTP for real-time event handling.

Methodologies

Response time

We utilized Nginx as the load balancer sitting in front of the provided web services. Whenever a client requests for web service computations, Nginx service will reverse-proxy the HTTP request to upstream web service containers, and also record the HTTP response time. The response time records will be real-time updated in a log file in the Nginx container volume, which we will use in the autoscaler program.

Replica/Service availability

Our autoscaler program is deployed as a single Docker service running in the swarm manager machine, which will periodically read from the HTTP response time and make judgements on whether to scale the web service up or down. To interact with the Docker swarm, we utilized the Docker Python SDK library, to call Docker Remote APIs for scaling operations.

The reasons behind a load balancer + independent autoscaler program instead of a proxy solution is that we want to have minimal impact on the upstream web service in a way that recording response time is fast and non-intrusive. Nginx is a modern and mature solution to reverse proxy upstream services and has great functionalities to record request information such as latency. With Nginx we also simplify the work on autoscaler, which only needs to read response times from a file.

Monitoring and Manual Interactions

The autoscaler used Flask to run an HTTP service that allows external interactions to turn on or off the auto-scaling feature.
The autoscaler server serves an HTML page for monitoring the auto-scaling behaviors. This client-server architecture also makes use of Socket.IO to allow real-time updates about replica, response times, and workload.

Design explanation

High level architecture

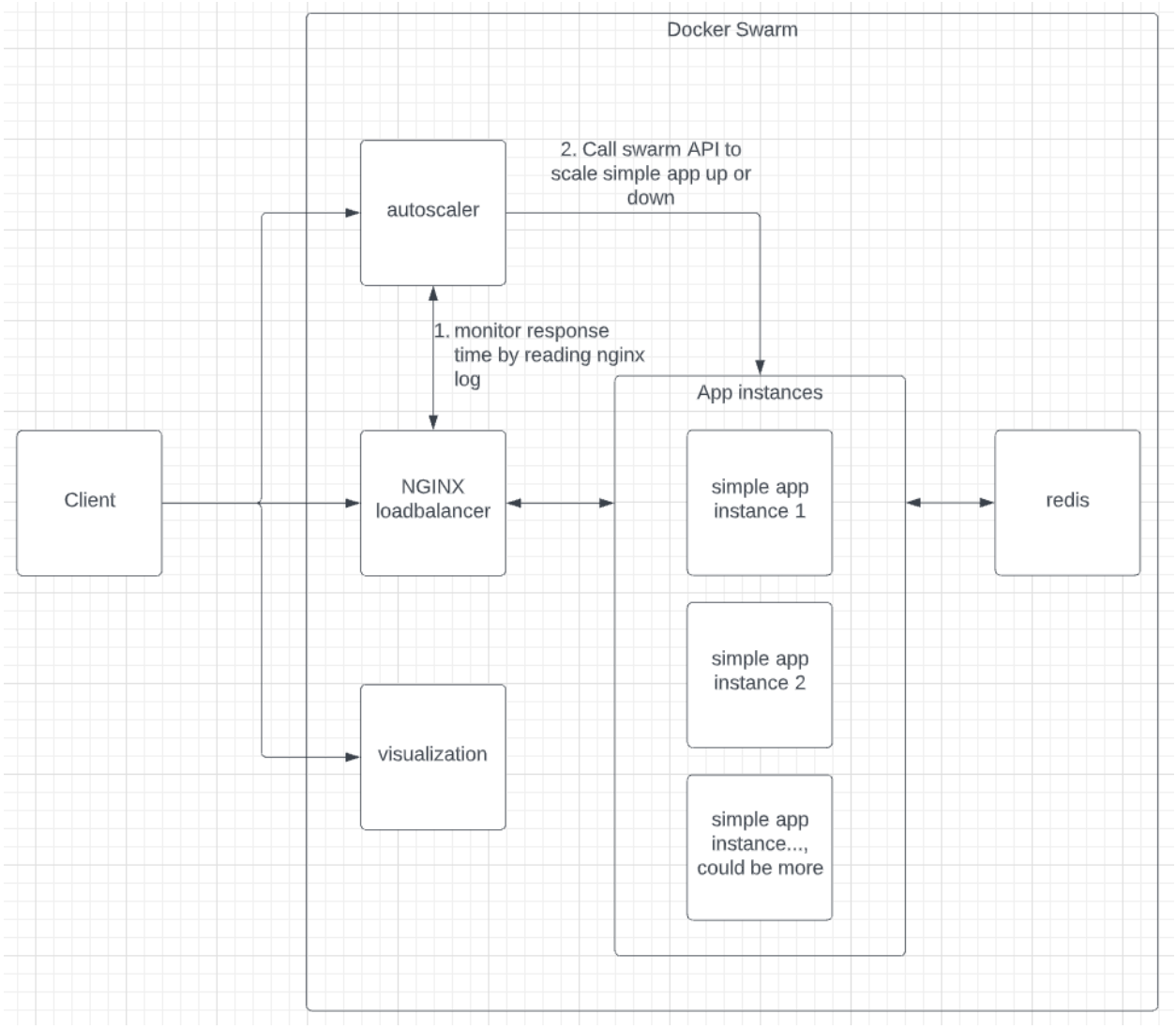

Figure 1. High level architecture

Describes the component of the project at a high level:

- Client: a user that can access the application through HTTP request. Using the browser, the client can send requests to access the visualizer, the graphs of workload and average response of the application, and toggle the auto-scale functionality.
- NGINX load balancer: a load balancer that routes incoming requests to different hosts within the docker swarm. NGINX also records the log time and response time of each request.

- Autoscaler: a custom python program that monitors the NGINX logs and scales up/down the web application using Docker Remote API depending on the average response time calculated.
- Visualization: a visual representation of the services within the docker swarm.
- App instances: consist of all the instances of the provided web application. The number of app instances will be scaled up/down depending on the workload of the application.
- Redis: a database to record dummy data.

State diagram

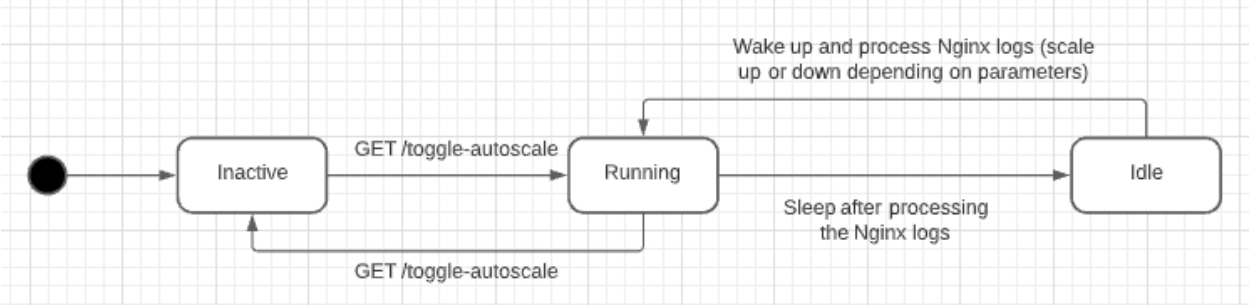

Figure 2. State diagram

Describes the states of the autoscaler. The autoscaler starts inactive until the user sends a GET request to /toggle-autoscale. Once toggled to active, the program will record the start time and sleep, transitioning to idle state. During idle, the program will wake up depending on the PERIOD parameter (default = 20 seconds). After waking up, the program performs the auto scaling algorithm and goes back to sleep. The cycle keeps repeating as long as the program runs.

Pseudocode of autoscaler

```
period = 20 seconds
lower_threshold = 2 seconds
upper_threshold = 5 seconds

while True:
      start = get_time()
      sleep(period)
      if active:
            f = open(nginx_log_path)
            times = []

            for record in f:
                  if record.time > start:
                        times.push(record.response_time)
            average_time = average(times)
            if average_time > upper_threshold:
                  scale_up()
            else if average_time < lower_threshold:
                  scale_down()

// nginx updates log files for each requests and record their response
time, please refer to the project/nginx/nginx.conf for log_format
// below is a pseudo-behavior for nginx updating the log file
while True:
      handle_requests()
      record = measure_response_time()
      write_to_log_file(record, nginx_log_path)

====================
Period is set to 20 seconds to avoid the ping-pong effect.

Lower_threshold is set to 2 seconds as that is the observed average
response time from a single instance under relatively low workload.

Upper_threshold is set to 5 seconds because we observed a linear
relationship between response time and workload for a single instance,
so with 5 seconds of response, on average, the instance is overloaded
with more than 2 concurrent requests, therefore we scale up.
```

Figure 3. Pseudocode for autoscaler

Once active, the program records the start time and goes to sleep for a period of time. Once awake, the program will read from the NGINX log and retrieve all the logs accumulated since before it slept. The program will calculate the average response time and scale the web application up/down. Finally the cycle restarts and the program goes back to sleep.

Deployment instruction

Environment requirements:

- Linux/unix machine (assuming Ubuntu 16.04)
- Python 3.5 installed
- docker installed: `sudo apt-get update && sudo apt-get install docker.io`

Deploy:

- Network security rules: on cybera, open up 5000, 8000, 8080 ports for TCP Ingress access.
- Start Docker swarm on manager machine: `sudo docker swarm init`
- On a worker machine, join the swarm: `docker swarm join –token <token> <manager_ip>:2377`
- Build autoscaler images and deploy the entire stack by running the prepared deploy.sh: `bash ./deploy.sh`

User guide

Once the stack is deployed:

To access visualizer:

- Go to <manager_ip>:5000

To toggle autoscaler active:

- Go to <manager_ip>:8080/toggle-autoscale

To see the graphs of workload, average response time, and number of instances:

- Go to <manager_ip>:8080

To access the simple web service:

- Go to <manager_ip>:8000

Conclusion

The auto-scaling for web services project provides hands-on experience with Docker and auto-scaling knowledge. By working on this project, we gained perspective on how auto-scaling can be leveraged to increase performance depending on the workload. Through simple auto-scaling by reading the NGINX logs, we see the average response

time drop from > 20 seconds to 2 seconds once the application is scaled up during heavy workloads.

References

[1]"Getting started with swarm mode", Docker Documentation, 2022. [Online]. Available: https://docs.docker.com/engine/swarm/swarm-tutorial/. [Accessed: 29- Mar- 2022].

[2]"Welcome to Flask — Flask Documentation (2.1.x)", Flask.palletsprojects.com, 2022. [Online]. Available: https://flask.palletsprojects.com/en/2.1.x/. [Accessed: 29- Mar- 2022].

[3]"Introduction | Socket.IO", *Socket.io*, 2022. [Online]. Available: https://socket.io/docs/v4/. [Accessed: 29- Mar- 2022].

[4]"Chart.js | Chart.js", *Chartjs.org*, 2022. [Online]. Available: https://www.chartjs.org/docs/latest/. [Accessed: 29- Mar- 2022].