Auto-Scaling Project

ECE 422

Authors:

Rowan Tilroe - 1629172
Sean Rutherford - 1516423

Abstract

This report describes the design process behind creating an autoscaler for a web application deployed using docker. The python tool locust not only simulates users accessing the web microservices, but its web UI provides graphics for response times and a way to manually increase user load on the microservices. By sending response data from locust over to the docker swarm manager, we were able to programmatically adjust the number of web microservices to keep the response time within a desired range. Overall, this project showed us how performability, a facet of reliability, can be achieved by dynamically increasing and decreasing resources.

Introduction

This report outlines the requirements, technologies used, architecture, deployment, and user guide for our implementation of the Reliability Auto-Scaling for Cloud Microservices project. The deployment section of this report describes the process of deploying a basic web application through docker that performs a difficult mathematical mathematical calculation when a user visits the website. With many users accessing this web app, the response time can become slow. Thus, we were to implement a way to scale the amount of web apps to guarantee the response time is within a certain range for “users”. If response times are low, scale down the number of docker web app services, while if response times are high, scale up the number of docker web app services.

Technologies

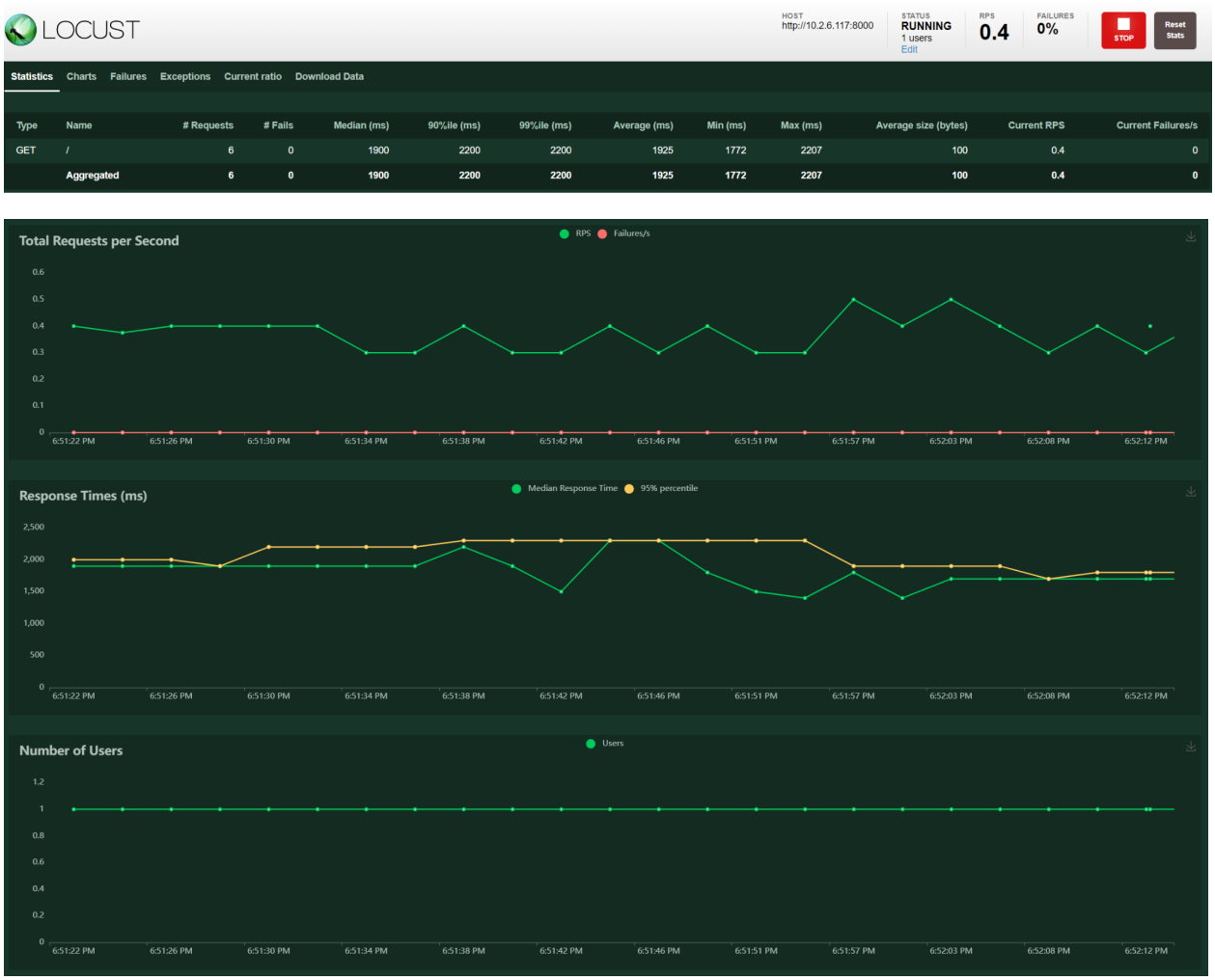
Docker

Docker is the platform that allows us to run services in small self-contained containers. For this project, we have 3 different services:

1. The webservice is a small web application that performs a difficult math task when a user accesses it
2. The datastore service (Redis) stores how many math tasks have been performed, and is not relevant to the scaling.
3. The visualizer service shows microservices are running on which VM. There are two VMs running docker microservices, a manager node and a worker node. It is not relevant to the scaling, but can be used to see how many web services are running.

Locust

Locust is a python load testing tool, which can be used to simulate users accessing websites and monitoring response times, among many other statistics. Locust comes with a web UI, which shows graphs about the number of simulated users, response times, number of requests per second. The number of “users” can be manually adjusted.



Python

The autoscaling portion as well as the locust workload simulating portion of the project were done in Python. Additionally, the docker engine was interacted with programmatically through python on the swarm manager VM. A few key python libraries were used to accomplish this.

- **docker**
    - The *docker* module provides programmatic access to the docker engine running the web microservice. Through it, we can change the number of web microservices running.
- **locust**
    - The *locust* module is the only medium through which the above tools mentioned in the Locust section are accessed. It was largely used to simulate users

accessing the web application through the swarm manager, as well as getting access to the average response times.

- **gevent**
    - The *gevent* module was used to create a side process on the client VM, which periodically runs the code used to get the average response time.
- **socket**
    - The *socket* module enabled communication between the client VM and swarm manager VM. The client VM sent over the average response time every 30 seconds to the swarm manager, which would adjust the number of web microservices.

Design Artifacts

High Level System Architecture

The high level system architecture illustrates which major components of the project are run where, and the communication between these components. In brief, the system can be roughly divided into two: The components on the Client VM side, and the components on the Swarm Manager Side. On the Client VM side, we have locust simulating users visiting the web microservice on the Swarm Manager / Swarm Worker, and communicating the average response time over to the Swarm Manager. On the Swarm Manager side, it accepts the average response time, and decides how to change the number of microservices, if at all.

Autoscaler State Diagram

The autoscaler state diagram shows a simple representation of the states that occur on the Swarm Manager autoscaler side of the system. The diagram does not have an end state as it is intended to run for as long as the user wants it to. The specifics of adjusting the number of microservices are shown in the autoscaling pseudocode design artifact.

Autoscaling Pseudocode

The autoscaling pseudocode graphically illustrates how the number of web microservices is adjusted based on the average response time. Brief justification for the important parameters is given as well.

Prerequisites / Deployment Instructions

The initial steps for setting up the docker microservices can be found in the project's Readme file up to step 7. These steps need only be performed once as long as the system running the docker services stays on.

Afterwards, the docker and locust python modules must then be installed. Be careful that the locust binary file is somewhere in a path found in the system's PATH environment variable so that the "locust" command is recognized.

Once all prerequisites are met, navigate to the project directory in both the Client VM and Swarm Manager VM. On the Client VM, locust can be started by typing "*locust*", and the web UI by going to localhost:8089 on your browser. On the Swarm Manager VM, the autoscaler can be started by running "*python3 autoscaler.py*"

User Guide

With both locust and the autoscaler running, the main point of interactivity for the user is on the locust web UI. The user can view the response time in both the "Statistics" tab and "Charts" tab. The user may change the number of simulated users and the rate at which they spawn by clicking the "Edit" button.

The user can view how many web microservices there are and on which VM they are running by going to the Swarm manager's IP on port 5000 in your browser.

Conclusion

This report describes the technologies and techniques used to implement an autoscaler for a web microservice deployed using docker. The autoscaler adjusts the number of web microservices to keep online to maintain a stable response time for simulated users.

The design artifacts provide a more graphical representation of how the overall system works. The high level system architecture shows how the system can be understood by dividing into the VM Client side, which simulates traffic, and the Swarm side, which runs and adjusts the number of web microservices. The autoscaler state diagram shows the states that the

autoscaler portion of the system goes through. The autoscaling pseudocode shows how the autoscaler decides how much to adjust the number of microservices, and provides some justification for important system parameters.

The prerequisites and deployment instructions provide the steps needed to run the system should a user want to run this system for themselves.