Cloud-Based Linux Server Performance Remote Dashboard Project

https://github.com/mk3justis/CBLSPRD

03/16/2025

COP3604.01

Justis Nazirbage

Introduction

The goal of this project was to build a GCP-deployed Linux-based server that would be put under stress tests and monitored remotely via a local application. This project served as a comprehensive overview of the course content that we have learned this far, with a large software engineering aspect included. For the server, I was able to practice things such as Firewall rules, VPC setup, bash scripting, Linux tools, and Linux file documentation. Using this documentation, I pulled data to find metrics completely from scratch using certain /proc files. Then, I created an API to send this data to a frontend, where I can view the metrics via my remote dashboard.

System Architecture and Design

The system was designed back to front. In the backend, I deployed a Linux server in Google Cloud, using the Firewall and VPC tools available to me. I used my load script to create a systemd service, which made the load script run on boot. In my backend, I pulled and parsed data from the /proc directory. For the API, I used FastAPI. For the frontend, I used React. The frontend fetches the data in a JSON format and displays it in charts available to the user.

Implementation Details

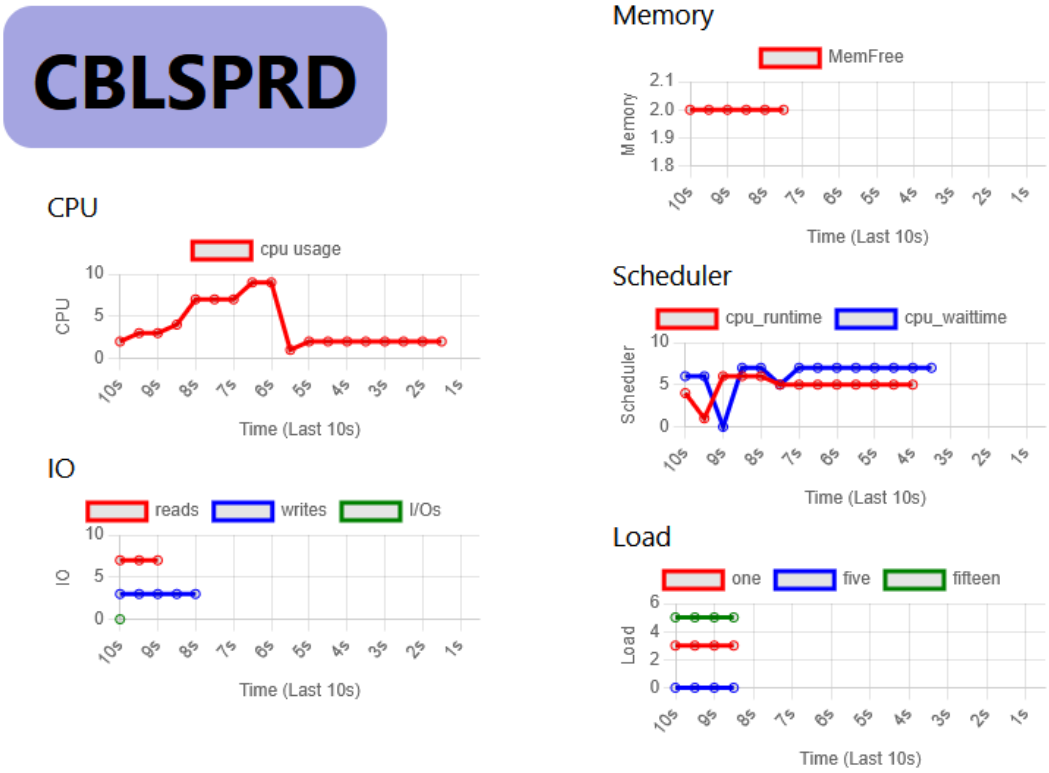
For the backend, there were a couple of major components that were developed. The first was the load script. This script was written to place a simulated load on the system using stress-ng. I wanted to vary the stress tests as much as possible with random tests for random intervals. To do this, I chose five classes available to stress as part of stress-ng's options. Then, for each of those classes, I found which stressors apply. To begin the stress-testing, I randomly choose a class. Then, I randomly choose a time for that class's stress to run. Then, I randomly choose stressors to run for five seconds each until the time for class stress has ended. The output of this can be seen in /var/log/load.log. A systemd service was created to run the service on boot.

The next major component was the FastAPI API. First I created a class for each of the metrics that I would use. In each class, I created a custom parsing function depending on the format, contents, and documentation of each of the /proc files that I wanted to use. For example, I used /proc/diskstats for my /io endpoint, and diskstats had many fields of information. However, I was only interested in the number of reads, writes, and IOs currently in use. After all of the parsing functions were done, I created all of the endpoints

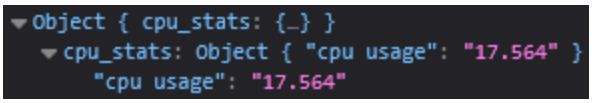
for each metric. Next, I created a cron job to check on the status of the API file to make sure that the API file would run again if it was killed. The cron job didn't seem to work, so I created a systemd file to restart it if it is killed. In my demo, stopping it normally doesn't work to showcase the restart, so I had to kill the process.

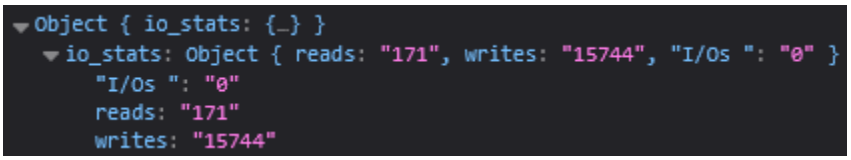
For the frontend, the main goal was to show the data that was fetched from the API in some graphs that are easy for the user to look at, compared to the vast amount of information available in the /proc directory that is difficult to navigate without reading the documentation. I used components to be able to reuse functionality of the graphs. This This allowed me to make each graph fetch its own data. However, because of this reusability, the graphs each had different data coming in, and differing number of fields based on what I felt was important. For example, I managed to condense the /proc/stat fields into one "cpu usage field", but I still had three fields for the /proc/loadavg file. I had to figure out how to dynamically bring in the values. This became a struggle when it came to the scaling for the y-axis, since the numbers shown do not reflect the data that is passed in.

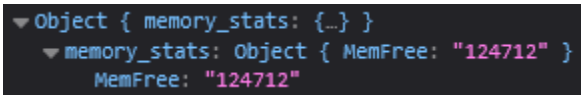
Experimental Results and Analysis

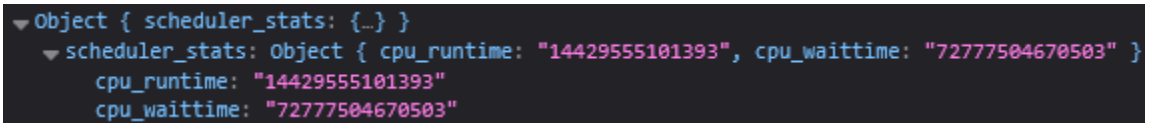


These screenshots show the graph for each of the five metrics that I chose to display using information from the /proc directory.









These screenshots show the information that is sent from the API after being parsed from the raw /proc files.

Problems Faced

There were several issues faced along the way. The first big problem was my interpretation of the instructions. I interpreted the instructions to mean that we had to find our metrics completely from scratch without using any Linux tools that are normally used for viewing metrics. This set me back as I read through many of the /proc files' documentation to see which ones would be useful, and how I would parse them to show meaningful data. This became an issue since the data I pulled was difficult to make sense of in the context of a remote monitoring solution, but my best effort was put in to research and utilize the /proc files.

Despite this setback, I feel like I learned about so much data that is available about the system in /proc. This made it worth it, despite my solution being imperfect in the end. The big issue that this misunderstanding caused for me was that my graphs have unmeaningful scales since it was difficult to translate the data into user-friendly data. I managed to convert the /proc/stat data into useful "cpu usage" data and graph it, but my scale was still showing up wrong compared to the data that was fetched.

If I could do this project over again, I would not do the frontend in React. It took up too much of my time, and there are better options for a small-scale project as this one. React does not have the best dependencies available for rendering dynamically changing graphs, compared to something like tKinter or plotly. The graphs update in realtime, but for some reason the scales are misleading, and the newest point is rendered at a seemingly random x–coordinate instead of the last x–coordinate of "1s".

There is a critical issue I have faced where the graphs will not render unless I manually render them first by initializing the graphs with random data. Then, I have to comment out that code and run the code that uses actual data. It will work without errors once I do that. I tried to debug it for many hours, but to no avail, and I couldn't figure out how to automatically initialize the graphs before running the rest of my code.

What Was Learned

From this project I learned and reinforced many concepts. I learned more about different ways to use systemd as a result of my cron job seemingly not working. I learned more about /proc files and how to read them. I learned about API development and React development. I was able to practice setting up the Firewall rule and the VPC. I also got to practice assigning a user and a group to the API.

Individual Contribution Table

Name	**Effort**
Justis Nazirbage	100%

Conclusions and Future Work

This project was a great learning experience for so many areas of my software-engineering skillset. That was the first API I developed without a team to assist me, and I learned a lot about React development from a variety of online resources. These two elements of the project were my favorite since I have been wanting to learn them, but did not expect to be able to incorporate them into any school project. It was a great way to tie together what I have learned in this class with the skills I learned about building software throughout my degree. I am happy that I worked individually on this project so that I could be "in the mud" with every component.

In the future, I would like to redo the dashboard with a simpler tool to hopefully fix everything that is not perfect with the current implementation. I would also like to try to implement the notification system. Aside from that, there are so many ways to build off of the ideas learned in this project. Between cloud deployment, remote communication, security features, backend development, and frontend development, this project can scale infinitely. I am looking forward to doing another Linux-based server project in the future.