CS 130 Hub

Assignment 8

This assignment has three parts: making or verifying your server's multi-threaded capabilities, setting up monitoring and a dashboard for your server, and then choosing, determining the requirements for, and designing the technical solution for a feature that you will implement in assignment 9.

Each student should submit this assignment by 11:59PM on May 30, 2023 into the submission form.

TABLE OF CONTENTS

Deploying CRUD

Deploy your server with a functional CRUD API handler. In order for your data to persist across container and VM instance starts, you will need to allocate a new Persistent Disk in Compute Engine, attach it to your VM, and mount it to your container to make it available to your server at the configured `data_path`.

Persistent storage on Compute Engine

To persist data across container restarts on Compute Engine, you will need to create a Persistent Disk and attach it to your Compute Engine instance, mapped into your container, as follows:

- Navigate to the VM Instances page and select the checkbox next to your instance, and click **Stop** to stop the instance. Once your instance is stopped, click on the instance name to go to the instance details page.
- Click **Edit** to go to the instance editor and then click **Add new disk** under **Additional disks**.
- Choose an appropriate **Name**, such as *webserver-storage*.
- Keep **Disk type** as *Balanced persistent disk* and change the **Size** to something smaller, such as *10 GB*.
- Click **Save** to create the new disk and return to the instance editor.
- Find the **Container** section and click **Change**.
- Under **Volume mounts** click **Add volume**.
- Change the **Volume type** to **Disk**, then choose a **Mount path**, such as */mnt/storage*. Choose your disk name (e.g. *webserver-storage*) for **Disk name**, then change the **Mode** to *Read/write*.
- Click **Done** to close the volume editor and then **Select** to close the **Configure container** editor.
- Click **Save** to save your VM instance settings, and then **Start / Resume** your instance.

Once the disk has been attached to your VM instance and Docker container, you may SSH to the instance and see the disk in two locations:

- `/mnt/disks/gce-containers-mounts/gce-persistent-disks/${DISK_NAME}`, e.g. `/mnt/disks/gce-containers-mounts/gce-persistent-disks/webserver-storage` from the VM instance native OS. If you need to upload or download data, you can use SFTP or SCP (see `gcloud compute scp --help`) to your VM instance to read or write files to that path.
- `/mnt/storage`, or whatever mount path you chose, from within the running Docker container. This is the path where your running webserver will be able to read and write files. To verify, you can run a shell from within your container and check.
    - SSH into your VM instance.
    - Run `docker ps` to find your container instance name (usually starts with `klt-`).
    - Run `docker exec -it klt-yourinstancename /bin/sh` to start a shell in your running container.

- `ls /mnt/storage` to see your files as viewed by your webserver process.

With the above Cloud configuration, your configured `data_path` could be a directory you create within `/mnt/storage`, e.g. `/mnt/storage/crud` (assuming you have created a `crud` directory).

Multi-threaded server

A web server needs to be able to handle requests from many users at the same time. For this part, you will modify your own web server so each request runs in a separate thread.

You should write a test (either unit or integration or both) to prove that the server can handle simultaneous requests. One way to accomplish this would be:

- Create a special blocking handler for a test path (e.g. `/sleep`) that blocks for a certain amount of time (i.e. `sleep()`)
- Launch a request to `/sleep` in one process
- Launch a request to another path in another process and expect an immediate response, verify the result came back in less than the time it took for your sleep amount.

Monitoring

In this section you will set up some basic monitoring for one or more of your server instances in GCP. We are interested in monitoring a few aspects of your server that will allow us to answer the following questions for any given point or range in time:

- Is your server running and serving requests?
- How many requests is your server serving?
- What kind of responses is your server issuing?
- Is your server currently being probed by a malicious botnet?

In order to answer these questions, we will use a combination of HTTP probing and logs analysis from Google Cloud services. You will need to write some code, as described below, and configure Google Cloud to set up monitoring and create a dashboard.

Create a health request handler

Your server should have a simple mechanism to report its status (even simpler than your status handler), such that a remote system can perform a health check on it and determine if the job is healthy, or if it needs to be terminated and restarted. The remote system should be able to request a path like `/health` and receive a positive response, a negative response, or no response at all if the server is really unhealthy.

To service this, you should create a new request handler that will always return a `200 OK` response, with a simple plain text payload of `OK`. Since your servers are still relatively simple at this point, we will assume that if they can return this simple response, they are considered OK. A more complex server might take into account many aspects of the system, or the state of some critical remote dependency systems (such as a DB being unreadable), and return a negative response.

This health request handler should be installed and configured to serve at `/health` in your server.

Handle malformed requests

To this point we have not explicitly defined behavior for responding to a malformed request. For your monitoring to be useful in detecting botnets and probers, you should detect malformed requests (such as ones that do not contain an HTTP request), and return a valid HTTP response with an error code. The “400 Bad Request” error code seems appropriate.

In other words, modify your server to return status code 400 on malformed requests if you do not do so already.

Add a machine-parsable log output

We have not been explicit about what your log output should be, so there is likely to be a wide variety of formats out there. Since log output is the easiest way of getting data into GCP’s monitoring system, you should add a semi-structured log line for each response your server returns that includes important information such as the response code. The goal should be to output a log line that can be easily searched for / match among all your other log lines (such as by containing some unique magic string like `[ResponseMetrics]`), and that contains metrics that can be matched and extracted with a regular expression.

Useful information you could include:

- Response code (required)
- Request path
- Request IP
- Matched request handler name

Extract metrics from your server

Once you have deployed your updated log output to GCP, you can start extracting your response metrics for further processing.

Start by navigating to Logs Explorer, where you should see all logs from all servers and services you are running. On line 1 of the text box under the **Query** tab, enter your magic string (`[ResponseMetrics]` or whatever you chose) and click **Run query** to scope the logs down to just the machine-parsable log lines. If you do not see any entries, verify that you have made some requests to your server after deploying and that some logs are being generated. To ensure that only logs from your servers on GCE are added, click the **Resource** dropdown above the query editor and select *VM Instance* followed by your instance name and zone, and click **Apply**, then **Run query** to update the *Query results*.

Once you are seeing only your machine-parsable log entries, we can create a metric to count how many responses your server has sent, broken down by response code. We will use a regular expression to parse the log lines and extract the response code. Start by clicking **Actions** at the top of the the *Query results* and select **Create metric**. Enter a **Log metric name** (e.g. *Responses*) and **Description** for your metric. Click **Add label** under **Labels** to add values to extract. For each piece of data you wish to extract, enter a **Label name** and **Description**, choose *STRING* as the **Label type**, and choose *jsonPayload.message* as the **Field name**. To actually extract a metric data from this line, you will enter **Regular expressions**. The regular expression must have a single group defined with parentheses, the contents of which will be your metric value. As an example, `response_code:([0-9]+)` would match a log line containing `response_code:200` and extract the value `200`. Verify that your regular expression works as intended by clicking **Preview**. Click **Done** to save your label.

Consider creating and extracting other labels for additional useful information output in your machine-parsable logs if you wish to breakdown and visualize your responses besides something other than response code.

You can leave the default **Units** empty, and **Type** set to *Counter*, then click **Create Metric** to finish creating your metric(s).

Create Uptime check

Next you will create an uptime check in the Monitoring portion of Google Cloud. Begin by going to Uptime checks in the Cloud Console.

Click **Create uptime check**. Keep *HTTP* for **Protocol**. Change **Resource Type** to *Instance*, and choose your main webserver **Instance**. Enter `/health` for **Path**. Click **Alert & Notification**. If you would like to create an alerting policy, you can have it send e-mail, SMS, or a Slack message when the check fails by editing the *Notification channels*, but we leave that up to you. Click **Review** and give your check a **Title**, then click **Test** to try your check. If it looks good (“Responded with 200 (OK)”), click **Create** to finalize your uptime check.

Create dashboard

Finally you will create a Dashboard from Monitoring. Click **Dashboards** in the left navigation of Monitoring, then **Create Dashboard**. Give your dashboard a name in the upper left of the Dashboard Editor that appears.

First you will create a chart showing the rate of various response codes given over time. Click **Add Chart** and select *Line*. Keep *VM Instance* for **Resource type** and then enter **Metric** and type *logging/user* to find your log-based metric for response code. By default, you may see many lines, one for each different response code, instance, path, IP address, etc. You can filter this down to a single GCE instance by clicking **Add filter**, selecting *instance_name* as **Label** and then the name of your main webserver instance as **Value**. Click **Done** to apply the filter. Then, to combine some of the lines together, click the box next to **Grouped**, then enter **Group by** and select your response code label. Change the grouping to **Sum** to count all the requests by response code, and you should see one line on the graph for each response code.

Next you should create some more charts. We suggest graphing metrics for:

- Response code (required)
  - See above
- Check passed (required)
  - Shows whether your server passed a check in a given minute. **Resource type** is *VM Instance* and **Metric** is *Check passed*. Add a **Filter** against the *name* label to match your server. **Preprocess** as *Fraction true*, with **Alignment function** of *max* and **Group by function** of *max* to ignore cases where not all prober locations failed to reach your server.
- Request latency
  - Shows the latency from the uptime probers to your server. Grouping by *checker_location* with function *mean* and using an aligner of *mean* seems to give good values.
- CPU utilization
  - Shows what fraction of an instance's CPU is being utilized.

Feel free to add any other metrics, and share your dashboard with your team via the URL. Take a screenshot of your dashboard once it has some useful information and include a link to your screenshot (perhaps share via Google Drive?) with the assignment submission.

Exhibit free will

For the final assignment, you will implement a roughly 1-week long feature or project on top of your existing server. Since the cross-team portion of the class is done, we (finally) have some room for teams and their servers to deviate from each other. For *this* assignment, you will do the majority of the preparation for next week's assignment, including choosing and specifying your feature, and carrying out the technical design work. You may also get a head start by starting implementation, if you are worried about your time to complete everything in the final week of instruction.

Choose a feature

Your team now has a C++ webserver capable of serving real files and executing configurable custom request handlers to arbitrary users on the Internet. Congratulations! Now, what can you do on top of that? We'd like you to think of how you could extend your server in an interesting way. To help you get started, here are a few ideas:

1. **Make your server deployment robust and scalable**
    - Set up a configurable prober or load tester. Find the limits of 1 instance of your server (number of requests per second, response time during load). Figure out how your web server fails under load. (Do you run out of memory? CPU?) Attempt to fix the bottlenecks. Investigate how the performance limit changes based on your deployment machine type.
    - Potentially set up Kubernetes on GCP (GKE) and set up a load balancer to auto-scale your instances and increase your deployment's capacity.
2. **Implement a small web application**
    - In the past we have had students implement a simple Meme Generator in a single assignment. You could take inspiration from this and do a variation, or do something functionally different but of similar magnitude and scope.
3. **Add support for Markdown**
    - Add the ability to render HTML from Markdown, which is commonly used to write and maintain simple documentation.
    - Author and serve a static documentation website for your server in Markdown (or any other website).
4. **Advanced HTTP**
    - Implement HTTP compression. One way to do this is to modify your code to allow chaining of request handlers, so that the compression handler can run on the results of the static file handler.
    - Implement keep-alive, so you can serve several requests over a single connection.
5. **Caching proxy**
    - This is similar to the reverse proxy, except subsequent requests for the same page should use the cached version.

- You should obey HTTP headers controlling cache behavior.
- For an extra challenge, make it so it can proxy any site, by changing the proxy setting in your browser.

6. **Support HTTPS**
   - Implement HTTPS encryption by using a library to handle the encryption.
   - You can get free SSL certificates from Let's Encrypt.
7. **Implement authentication**
   - You could implement HTTP authentication, which would require you to have a way to set and check usernames and passwords.
   - You could integrate with Google's OAuth system to process logins from UCLA addresses.
   - You could use authentication to protect against protected webpages, or do display information about a user in a returned webpage.

For each of these ideas, it is not expected that you would necessarily adhere to what is described above exactly. It is up to you to choose a direction, and create the full specification of the feature. Each of the above suggestions could be arbirarily simple or complex. It's up to your team to decide how far you wish to take it, and the more ambitious teams will get higher grades on this assignment. If your feature is too easy or simple, the graders will not be impressed. Be careful with your ambition though, because you also need to complete the next assignment! There's little value in being overly ambitious and then being unable to complete the assignment.

Write a PRD

Before a team writes a technical design document, someone (such as a Product Manager) should write a Product Requirements Document (PRD). The PRD should describe the project/feature/product in terms of the functionality without diving into the technical details. It is far less technically-oriented than the technical design document, and generally informs the requirements and functionality that a technical design will need to satisfy.

For this assignment, your team should document the choice of your project by writing a PRD. Please start with the template below:

Some Feature PRD

Team: *Your team name here*

Author(s): *Your person names here*

Vision

Very brief (1-2 sentence) summary of the envisioned product/feature.

Motivation

Why is this product/feature being requested and built? What about the current state of things is unacceptable, or needing to be improved? Consider including quantitative data/metrics if available.

Goals

What are the specific objectives? What are non-goals? Consider presenting this in bullet form.

Requirements

What will this feature need to do? How should it behave? Use strong declarative words like "will" and "shall," and avoid ambigious terms like "might," "could," and "may." Categorize and number them for reference. For example:

First category

1. **Requirement name:** Requirement description
2. **Requirement name:** Requirement description

Second category

1. **Requirement name:** Requirement description
2. **Requirement name:** Requirement description

Success criteria

How do you measure success? Include quantitive metrics if possible. Include qualitative measures such as newly enabled user experiences.

Write a Design Doc

With your product requirements finalized, you should write a technical design doc to describe your team's proposed technical implementation to the functionality described in the PRD. Similar to the *API Design Doc* from Assignment 5, your document should be sufficient to inform others about your technical intention in implementing or carrying out your feature, and should provide the base reference for any questions that arise during implementation.

Please start with the template below. Note that some of this is duplicated with the PRD, and that's OK. These documents are intended to serve different purposes, with different audiences, and are often written by separate people.

Some Feature Technical Design Proposal

Team: *Your team name here*

Author(s): *Your person names here*

Objective

What are you doing and why? What problem are you trying to solve? Include major goals and non-goals. Stay high-level, maximum 1 paragraph. Leave the details for later sections.

Background

What is the background of the problem? Where is this design intended to be implemented? Consider including links to your code repository or other documentation.

Requirements

What are the requirements that your design should meet? What are the detailed goals of the solution? What are the explicit non-goals that you are choosing to ignore? The guideposts of your problem as stated here should help inform the ***Detailed design*** *below. This may be a summary of the requirements in the PRD, with link(s) to relevant section(s) in the PRD.*

Detailed design

Here is where you describe your design in detail. Include any code snippets, example configurations, and other details here. Create sub-sections for each major component you are discussing.

Alternatives considered

*List any alternative design choices you considered, and justify why you ultimately did not choose them. Justifications may reference points in the **Requirements** section (e.g. failing to meet goals, or unnecessarily meeting non-goals).*

Grading Criteria

Team grading criteria include:

- Deployed CRUD handler
- Implementation and testing of multithreading
- Implementation of a health handler
- Logs that drive metrics (check passed, response code, 2 other charts)
- A dashboard that charts the metrics
- A PRD justifying a new feature
- A Design Document that describes and analyzes the trade-offs and engineering work to implement the PRD

Individual Contributor criteria includes:

- Code submitted for review (following existing style, readable, testable)
- with Reviews that address and resolve comments from the TL and other team members
- or Written work or other artifacts related to PRD/Design documents

Tech Lead criteria includes:

- An assignment tracker completed and kept up to date
- Comprehensive meeting notes
- Well formatted Product Requirements and Design documents

Submit your assignment

Everyone should fill out the submission form before 11:59PM on the due date. We will only review code that was submitted to the `main` branch of the team repository before the time of the TL's submission. You may update your submission by re-submitting the form, which will update the submission time considered for grading purposes if you are the TL.

Late hours will acrue based on the submission time, so TL’s should avoid re-submitting the form after the due date unless they want to use late hours.

“You may think using Google’s great, but I still think it’s terrible.” —Larry Page

Page last modified: May 23, 2023.