Entry Task for Golang

1. Requirement

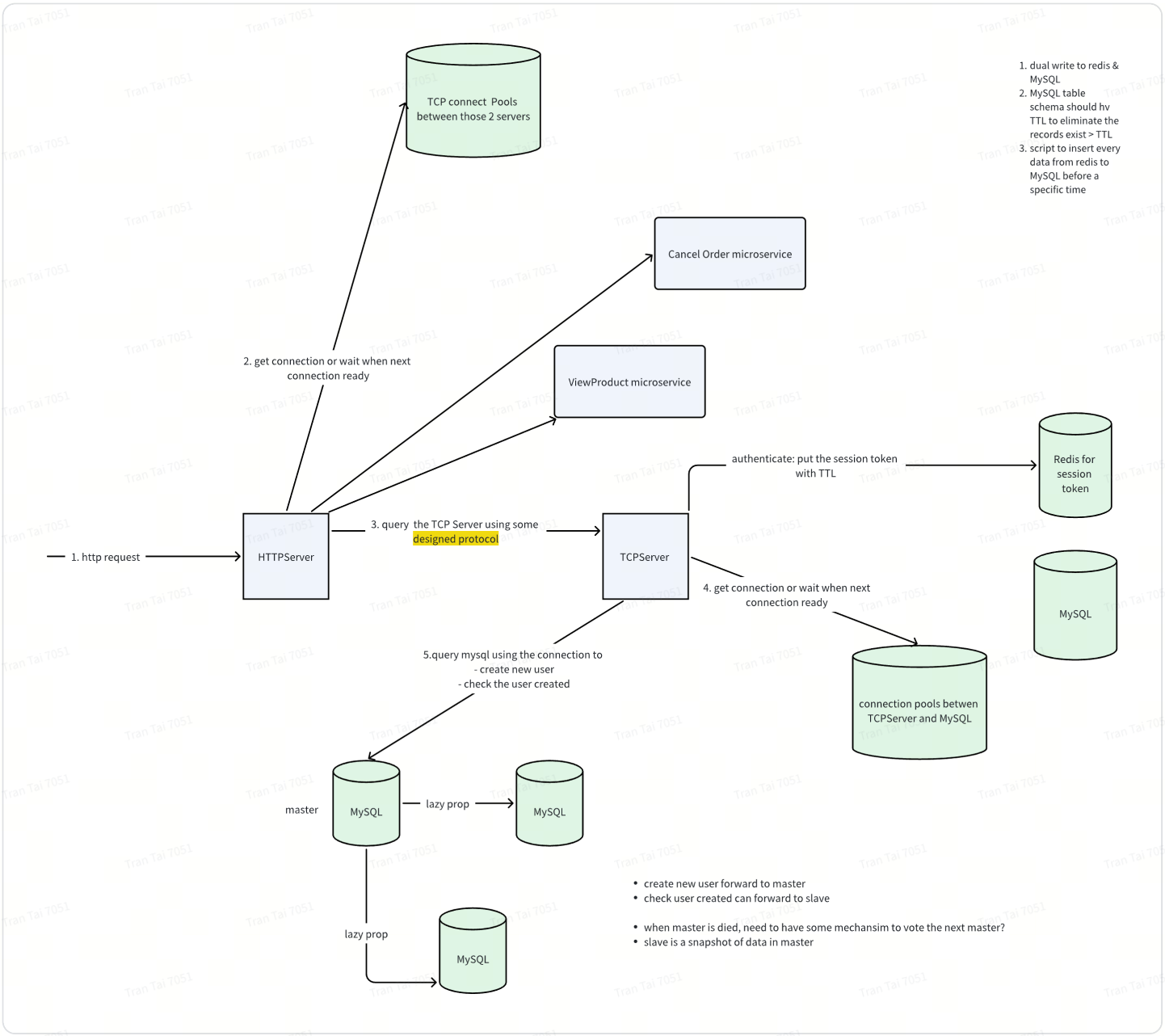
- Build 1 user signup and login system
  - For simplicity, assume the user will be registered/authenticate with our system using **username** and **password**.
  - Aft calling the login api, the user could be able to view the product.

- 1 Process/Container will act as an HTTP Server to accept requests from users.

- 1 Process/Container will act as a Signup/Login Microservice (a TCP server that access User MySQL DB)

- Performance must reach 1000 QPS and latency must be < 1s for each request.

2. Architecture



- TCP connection Pools:
  - Struct {

        sem Semaphore(N); for N nil connections;
  - }
  - acquire() {

        // Try to take one connection from the semaphore

        // if connection (conn) is nil, assigned conn = (new connection to the TCPServer).
  - }
  - release(){

        // release the connection back to the semaphore
  - }

- Background process to change to close the connection that exceeded its TTL.
  - This is not correct design for the connection in connection pool.

3. Details Design

3.1. API Design

- APIs will be designed with RPC format.

Name	Path	HTTP Method	Payload	Response
Register API	/open_api/v1/register	POST	`{ "username": "username", "password": "password" }`	[200] Success. [400] Bad request: the username, password not follow its naming convention. [409] Fail as there is an already registered user in the db. [500] Fail as there is another internal error.
Authenticate API	/open_api/v1/authenticate	POST	`{ "username": "username", "password": "password" }`	[200] Success, return a session token back to the client. Think where this token is being stored for later step. `{ "token": "122nfjkedvd" }`

				[400] Bad request: the username, password not follow its naming convention. [404] Fail as the user is not found in the db. [500] Fail as there is another internal error

- Storing the password directly in the database is not secure. Which **hash algorithm** should be used to do that?
  - Use a **modern hashing algorithm**.
    - The purpose is to make the resources needed to crack as intensively as possible (**slow functions**).
    - Other hashing algorithms like Md5 or SHA-1 are **fast function**

      which is not secure and can't be used.
    - However, one-way hashes with **pre-computation attacks**
      - Some common attacks like rainbow table`s, database-based lookup.
  - **Salt the password**
    - Salt is a **unique randomly generated string** that's added to each password as part of the hashing process. This will be **stored tgt with the hash value in db**.
    - When user login -> calculate the hash of the user password tgt with the salt in db and compare with the hash value in db.
  - **Peperring**:
    - Apply the 2nd layer on the salt password. The value stored in db will be the encrypted hash value by using a secret key.
    - This is shared between all stored passwords and not unique for each password, like salt.
    - This **must not be stored in db.** Instead considered stored in Hardware Security Modules.
    - Need to consider **rotation strategy like other cryptographic key.**
  - **Work factor**

        - No of iterations the hashing algor performed for each password. The value is often (2^k).
        - This is often **stored in the hash output**.
        - Strive between balance of security and performance.

    - Will spend time diving deeper later but not my current priority now.

      https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

- How to generate sessionToken?
    - HMAC + SHA256

  Hash-base message authenticate code
- How to generate id unique in distributed system:
    - Uuid: https://www.linkedin.com/pulse/what-exactly-uuids-how-sounique-prashant-pandey/
    - Twitter SnowFlake approach

different sections. Figure 7-5 shows the layout of a 64-bit ID.

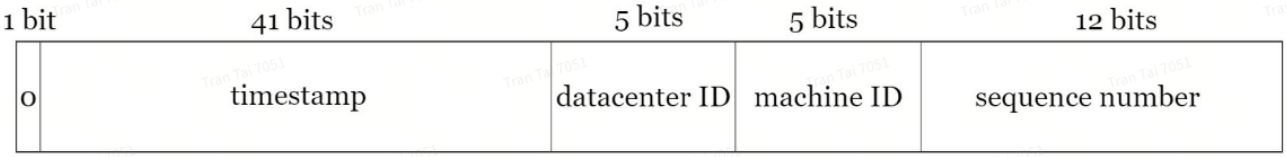


Figure 7-5

3.2. Protocol Design between 2 servers

Request

- The data sent via the connection will be in this format:

```
[COMMAND_LEN] [COMMAND] [username_len] [USERNAME] [password_len] [PASSWORD] [name_len][NAME][AGE]
```

- COMMAND can be REG | AUTH. Command len will then be 3 and 4 respectively.

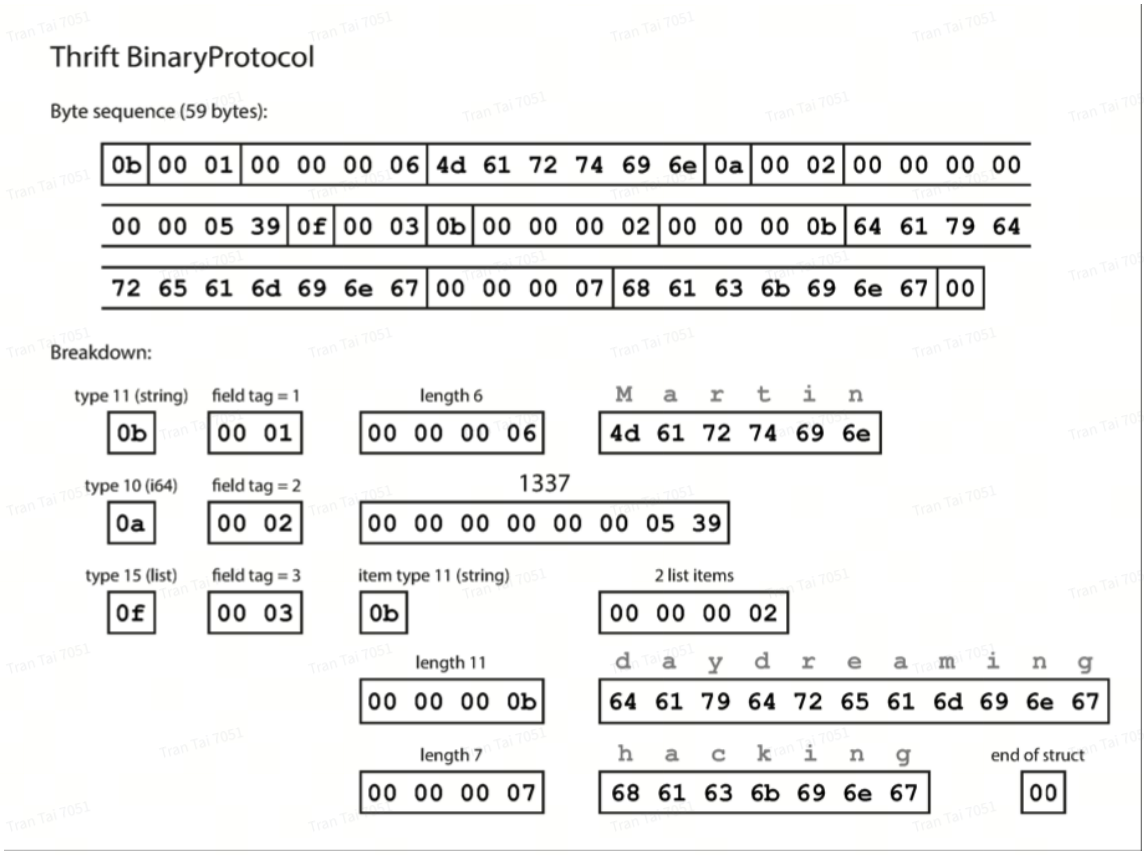
Response

```
[STATUS][err_message_len][ERROR_MESSAGE]
```

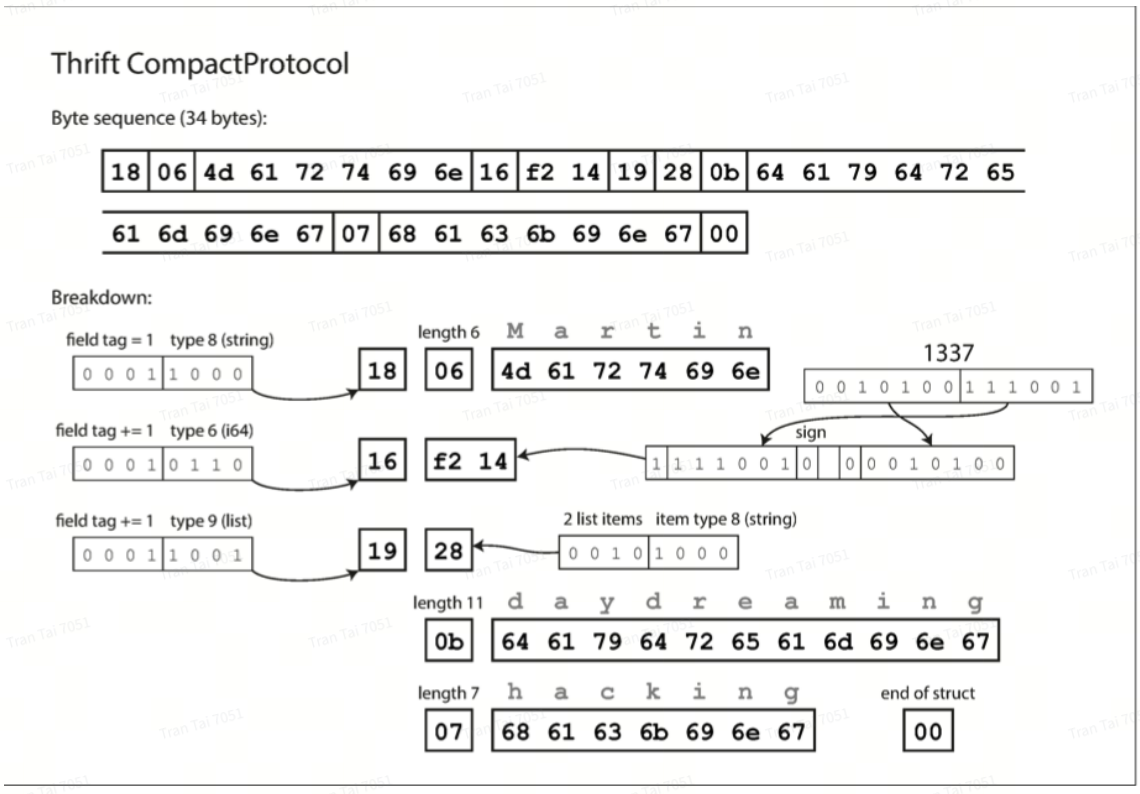
- When the client writes to the connection but the server does not read all the bytes, what will happen?
  - The write operation from client will be blocked.
- How to make sure compatibility (When request format change + regardless of deployment order of 2 servers).
- How to make sure the protocol design has good performance.

  Some notes from Data Encoding:

  - Thrift and Protocol Buffers:
    - They are binary encoding libraries which come with a code generation tool that takes schema definition like the ones above.
    - Thrift has 2 different binary encoding formats: **BinaryProtocol** and **CompactProtocol**.
      - **BinaryProtocol**: almost similar to MessagePack encoding. (each field has annotation for its type + length annotation for data length + data). The difference is that there are **no field names**. Instead, encoded data contains **field tags** (number).
        - (type (1 byte) + field tag (2 bytes) + data)



- **CompactProtocol**: it packs the field type and field tag to 2 bytes and uses variable length integer with the **top bit of each byte to indicate whether there are still more bytes to come**.
  - ((type + field) (2 bytes) + data with variable length)



- Has dedicated list datatype then can possibly support nested list.

- Protocol Buffer: has encoding design almost similar to Thift Compact Protocol.

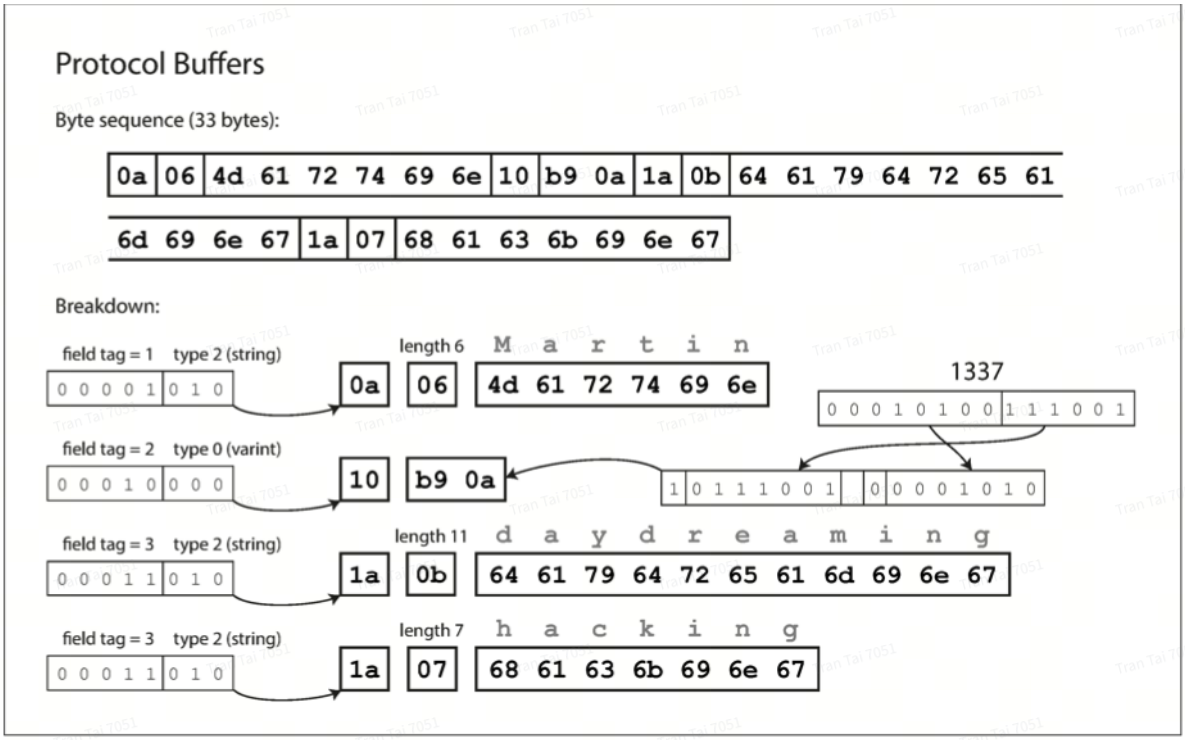
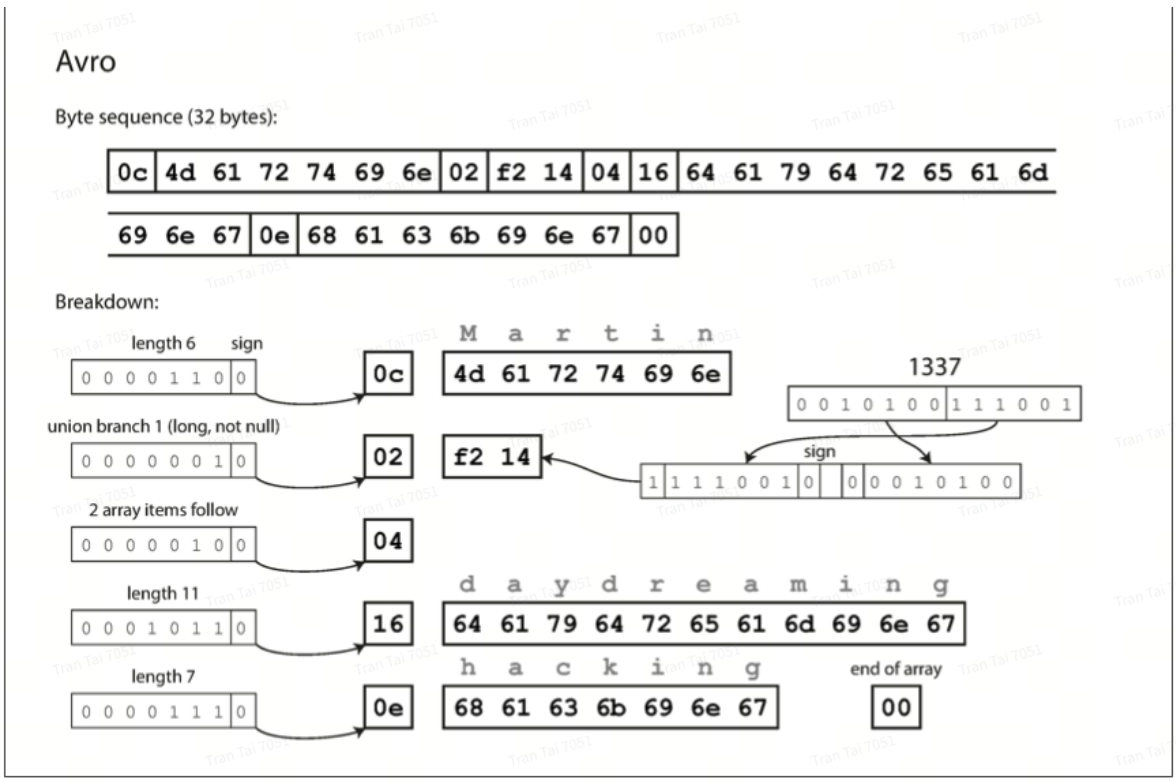


Figure 4-4. Example record encoded using Protocol Buffers.

  - **repeated** as a marker for field means for the same field tag can appear multiple times in. The record.

- Avro:



  - Key idea: write's schema and reader's schema no need to be the same, just need to be compatible.

        - When data read, Avro library resolves the different by looking at the write/reader's schema and translate data from writer's schema to reader's schema.
        - If want a field to be null, should use **union type.**
        - One advantage of Avro is the schema does not need storing tag numbers.

    // TODO: redesign the schema here .

3.3. Database Schema Design

- User Table DDL:

```
1 CREATE TABLE `user` (username string, password string) INDEX
  name_password(username, password);
```

- Session token:
    - Table schema to store in MySQL and Redis.

4. Need to take a looks

- How load balancer works?

  What is load balancer

  - Evenly distributed the traffic across multiple servers.
  - It is a intermediary between clients and servers.
  - Operate on **network layer** (focus on routing based on IP addresses and ports) and **application layers** (e.g. HTTP headers and cookies)

  Algorithms:

  - The 2 types of algorithms use:
    - Static: Distribute the requests to server **without** caring about the **server real time conditions** and **performance metrics**.
      - Pros: Simple to implement.
      - Cons: less adaptive.

        - Some algos:
            - Round Robin: Normal version, Sticky version and Weighted version.
            - Hash-Based: hash the IP ->map to the servers. Consider **Consistent Hash**.
    - Dynamic: Take in consideration of server metrics and conditions.
        - Some algos:
            - Least connections:
                - route request to the server with least connections.
                - need to keep track of how many conns to each server.
            - Least Time:
                - Send request to server with lowest current latency/fatest response time.
    - Need to consider between performance, constraints and capabilities.

- From http server use TCP con to multiple TCP server. Can load balancer really works?

- When a container is removed/added for a server, how to let the load balancer aware?

- How to handle the previous question with N load balancer? What happened if 1 load balancer died?

- When getting data from slaves, there might be some delay with the changes from master. How to handle this?

5. Tools and Takeaways

- How to build a TCP/HTTP Server in Golang.
- Wrk, k6s: to measure QPS to the server.
- PProf to do profiling.
- Learn k8s and containerizations to scale those servers.