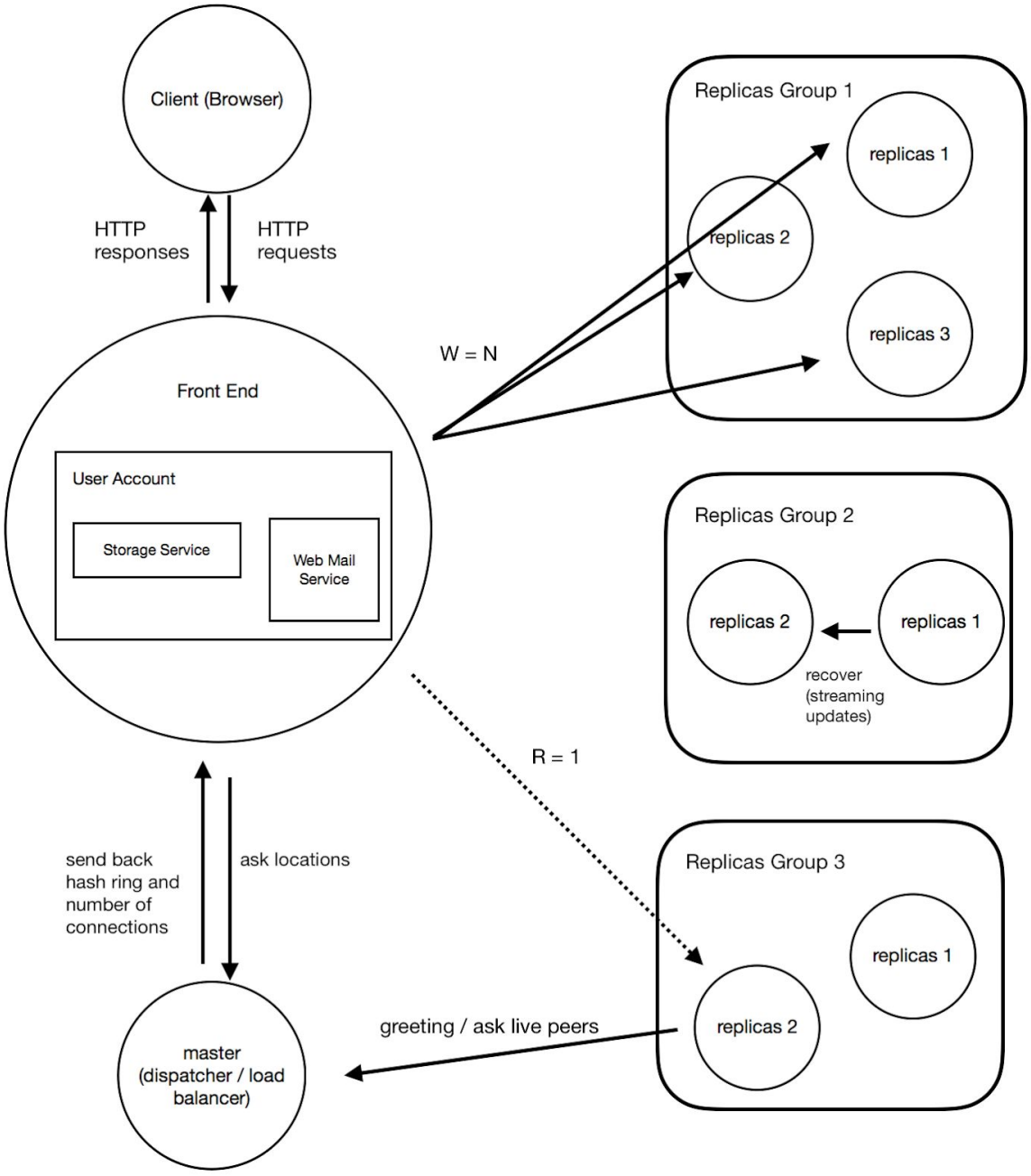
T22 CIS-505 Final Report

Team members:

- Yizhou Feng: Frontend Server & User Account
- Shuang Liu: Frontend Server & Admin console
- Kaixiang Miao: Storage System && Storage Service
- Shuangchen Shen: Webmail Service

Diagram



Components

- HTTP Server
- Web Mail Service
- Storage Service
- Key-Value Storage System

Supported Features

- HTTP Server
    - Handle GET/POST/HEAD HTTP requests
    - Cookie handling
- User Account
    - Register new account
    - Change password
    - Login / check password
- Web Mail Service
    - Compose new Mails (send to both internal and external users)
    - Receive Mails (from internal users only)
    - Reply Mails
    - Forward Mails
    - Delete Mails
    - Display the list of all mails
    - Display the title, sender and recipient name, mail content of each mail
- Storage Service
    - Create Folders
    - Upload Files
    - Delete Files/Folders
    - Move Files/Folders
    - Rename Files/Folders
- Key-Value Storage System
    - GET
    - PUT
    - CPUT
    - DELETE

Architectures and Design

Overall

Communications between the front-end services and the key-value storage system are done by RPC(gRPC). The corresponding services will be called based on the URI of the accepted HTTP requests.

HTTP Server

When each HTTP request is sent, the server parses each request line and parses HTTP headers for each request line. Each message with a header would be stored into a request's corresponding field. Upon matching the uri, the Get method would response a same message body. The Post method would seek for post data, and use the post data to make connection with backend to do relevant tasks. The Head method works similar to the Get method, except that it does not handle message body.

For cookie handling, cookies are gotten from usernames and their password as session ids. It then stores the session id, paired with its username to the backend. Cookies/ session ids are parsed from "Cookie" header. When a user is back to the web page again, it checks if the user is paired with a session id in the backend. If it is, the frontend server directs to the user information page.

User Account

When a user does register/login/change password operations, the username and its password are get parsed as post data from an HTTP request. For register, it checks if "password" matches "confirm password" and checks if the username is already exist. If both answers are not, then store it to backend. For login, it checks if the pair of username and password is existed in backend database, and determine whether allows the user to login. When a user is already logged in, he can change his password. If all fields are matched, it will update the username and password in the backend database.

Web Mail Service

The overall process is like this: the browser first sends a request with corresponding action name, attribute value and current username to the mail server. We need to first parse the request, and then mail server sets up a socket connection with the corresponding back-end storage server and uses the protocol defined in the project description (PUT, GET...) to perform actions. After getting responses from its backend server, the mail server calls a function to create a dynamic html page and returns it to the browser for displaying.

Our webmail service provides both internal email transmission and external transmission. For the internal part, registered users can send/reply/forward emails to him/herself and also other registered users (e.g. from bob@localhost to ava@localhost). For the external part, the smtp client queries the DNS server about the mx records of the recipient's domain, and then the DNS server would send back a list of mx records. We then use DNS to get ip addresses of all mx records, and try to build a connection with one of the ip addresses. If the connection fails, try another one until it succeeds. We can successfully send emails to gmail users. The content of emails is decoded by Base64.

Receiving emails from outside-system users is not required, and we didn't implement this part. However, if we could have a public ip address which could be accessed by outside machines, then receiving emails from outside-system is not a very difficult task. To achieve this,

I am considering deploy our program to a cloud service, like Amazon. Maybe we can make that work in the future.

Storage Service

Each file/folder has an unique ID which is generated by the user name of its owner, the path and the current timestamp. The unique ID is used as a row key in our storage system. That row stores its file/folder information including the name of the file/folder, a tag representing if it's a folder, an unique ID of the next file/folder, an unique ID of the child file/folder (if the current row is a folder) and the size of the file.

A new table is generated when an user logins in from the web page. Each path is mapped to the ID of the file it points to. In this way, renaming and moving can be easily done by modifying the mapping information without actually changing lots of things.

Key-Value Storage System

The key-value storage system consists of multiple servers, including a single coordinator(master) node and many slaves. The master node is responsible for storing Each of these servers is event-driven in order to handle requests efficiently. The event-driven asynchronous architecture also allows us to avoid race conditions easily comparing to the multithreading architecture. For example, nothing special needs to be done to guarantee the atomicity of the row operations.

- Communication

Communications within our key-value storage system are also be done using RPC(gRPC). This is surprisingly useful when serializing the data to disk for checkpoint and message logging because the same protobuf data format provided by gRPC can be used for serialization to the disk as well. When a node is recovering and asking for another node for a checkpoint, the data on the node being asked can be read and sent over the network without any intermediate data format conversion.

- Partition

Consistent hashing is applied in our storage system. The hash space is 2^64. Each slave has its own hash key. An operation is dispatched to corresponding slaves based on the hash value of the row. When a new node joins, it's put in the middle of the longest part of the hash ring. The hash algorithm we use here is MurmurHash, which has a good performance and guarantees the hash value is evenly distributed.

- Load Balancing

Each node on the hash ring has some replicas. The master node maintains the number of connections of each of these replicas, which is sent along with the location information when an

user is issuing a request to our storage system. Requests will be forwarded to the replicas which have the least number of connections.

- Replication

The replication protocol in our key-value storage system is quorum-based protocol. We simplify the protocol a little bit. In our system, W is always the total number of the replications(N) and R is always 1. This approach comprises a bit of write performance, while read performance is maximized.

We simplify the protocol mainly because things are much more complicated when W is smaller than N. For example, consistency can not be guaranteed if the nodes with the latest data fail even if there're still more than R replicas alive and recovering is also not easy because it requires tracking the current quorum as well as the original quorum of the failed node.

- Consistency

All the operations(GET/PUT/CPUT/DELETE) are at least row atomic. No locking scheme is applied for GET/PUT/DELETE. However, atomicity between CPUTs is guaranteed. Hence, our system might become inconsistent if concurrent writes using PUT happens. Users are supposed to use CPUT instead of PUT if strong consistency is desired.

When a CPUT request is issued, it tries to acquire W locks first before starting writing to W replicas. If it can not acquire all these W locks, no write happens. This guarantees the atomicity of CPUT and also provides a strong consistency. To avoid deadlock, if a lock being requested is holding by others, it will drop all the locks it has acquired, wait for a random amount of time and retry again. This prevents deadlock but livelock might still happen. But in practice this approach works well if there aren't too many connections coming in.

- Fault Tolerance

Since W is always N and R is always 1, as long as there's one replicas alive, our system continues working as if no node fails.

- Recovering

Recovering is done by checkpoint and message logging. Each server periodically dumps its table to the disk, clears the logging file and starting recording the logging for write operations again.

When a failed node comes back, it starts listening the incoming messages and putting them into a event queue without actually applying them before being actually recovered. Since our servers are event driven, these messages are queued conveniently in the event queue.

Then it contacts the master node to see which peers are still alive and ask the live node for the version number of its checkpoint. If they have the same version number, the failed node is simply recovered by using its own local checkpoint and streams the updates from the live node

by using the logging of the live node. If they don’t have the same checkpoint version, it copies the checkpoint from the live node as well as its logging and then replays the logging updates. Once the failed node finishes replaying, it can continue handling the message events in the event queue and is finally alive again.