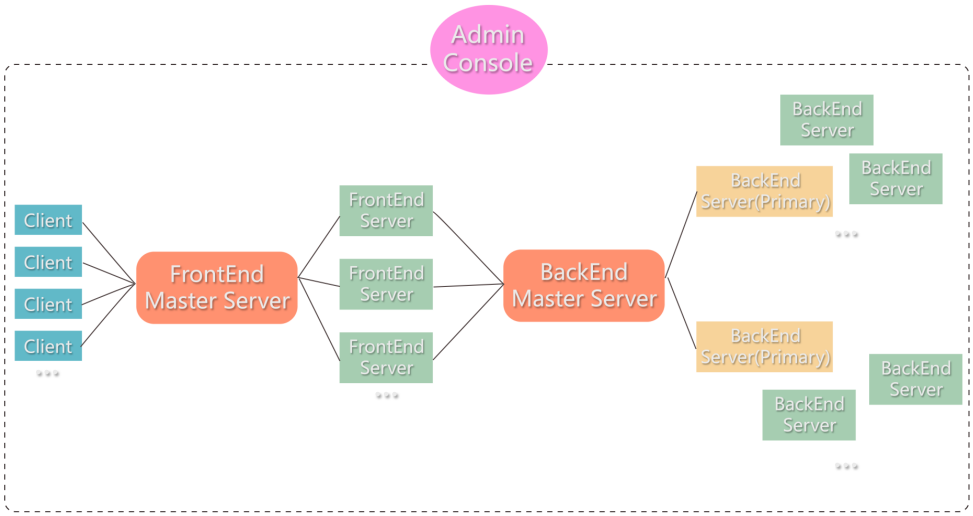
PennCloud Project Report - Team15

Overall Design



Generally, we implemented a small cloud platform in which users can register their own accounts, upload/download files, and send/receive emails between both local accounts and outside mailboxes.

Implemented Features

Backend

Distributed Key-Value Store

- ➢ Primitives: PUT(r, c, v); GET(r, c); CPUT(r, c, v1, v2); DELETE(r, c)
- ➢ Rows and Columns of the tables store information of files and support primary operations as mentioned above
- ➢ We choose tablet size to be 100MB so that it not only keeps good load-balancing but also requires less bookkeeping

Fault Tolerance and Recovery

The server will send heartbeats to the admin server and the load balancer so that they will know the status of each server. After a server is started from the terminal or awakened by the admin server, it will go through the following three stages:

SET stage (does not know a primary) : Server will send "SET" request to balancer when it starts, and wait for balancer return who is the primary. Note that if the current primary is down, server will receive a "SELECT" request from the balancer, then it will

also change to SET stage. Then, it sends a "SELECT" response to the balancer including its version information. After it receives a "RESUME" request from the server with the information of the new primary server, then it goes back to WORK stage and sends the "RESUME" response back to the balancer.

RECOVERY stage (knows primary and needs to load the ckpt and log): If the server itself is a primary, then it can just load the ckpt and replay log from the disk. If the server itself is not a primary, then it will send "RECOVERY" request to the primary about the latest log and ckpt version. If ckpt is not the latest one, the primary will need to send its ckpt to the server. If log is not the latest one, the primary will need to send its log to the server.

WORK stage: Only after a server knows the primary and successfully loads the tablet to the memory, it will start to parse new requests in the request queue.

Data Replication

At first, the frontend server will query the backend load balancer, and the balancer will return the primary of the less loaded replication group to it. Then, the frontend server will directly send its request to the primary as long as it is alive, and the primary will then forward this request to other nodes in its group to synchronize the replication.

Logging and checkpoint

To be able to recover after a server is shut down, we save two checkpoints periodically and keep updating a log every time we receive a request from the front end. We have:

- A meta file which contains the size of the value for each cell, the rowkey, colkey and version number of this row.
- A checkpoint file contains the rowkey, colkey and the value.
- A log file which contains all the requests sent after the last checkpoint is saved.

When a server tries to load a checkpoint, it will load the metafile first, and then it can use the size of the cell to decide when to stop parsing one cell content. Then, it will parse and replay the operations in the log.

Frontend

Load Balancer

- Load Balancer will redirect clients to one of the frontend servers.

HTTP Server

- Support communication between frontend and backend servers.
- Support multiple client connections as a multi-threaded server.

- ➢ Parse HTTP requests from frontend browsers to GET, HEAD, POST, OPTIONS, and other handler functions.
- ➢ Parse HTTP requests from frontend browsers to get HTML files.
- ➢ Retrieve all files of a user from backend servers to frontend clients.
- ➢ Retrieve all emails of a user from backend servers to frontend clients.
- ➢ Forward outside emails from frontend clients to SMTP clients.
- ➢ Receive in-system emails sent by Thunderbird from SMTP servers to backend storages.

User Account (api/account/)

- ➢ Login as an user already has an account
- ➢ Register for a new user
- ➢ Change password
- ➢ Token-based Authorization: when logging in, the user will get a unique user id and an encrypted access token from the HTTP server, and then the client sends api requests with the access token to the server, and thus the server knows which user is sending the request.

File (api/file/)

- ➢ Upload (text / binary) / download / delete / rename files
- ➢ Create / delete / rename folders
- ➢ Nested folder: when the user is at the root directory and wants to get files at the root directory, the url param will be *filePath = root/*, and the user will get all files at the root directory. If the user clicks a folder “CIS505”, the url param will be *filePath = root/CIS505/*, and the user will get all files at root/CIS505/ directory.
- ➢ Move file from one directory to another

Webmail Service (api/mail/)

- ➢ Receive & send emails
- ➢ Delete emails
- ➢ Reply emails
- ➢ Forward emails

SMTP

- ➢ SMTP Server: The SMTP server can accept emails from outside systems like Thunderbird, process emails and send those emails back to the frontend server. Then the frontend server will send them to the backend.
- ➢ SMTP Client: The SMTP client can receive emails from frontend server, looking up DNS records from domain name and get IP addresses from DNS records, then send emails to outside the system (SEAS email account)

Admin Console

Dashboard (api/admin/)

- ➢ Get all servers' status and key-value pairs
- ➢ Change server status (stop / restart)

Extra Credit Features

Authentication

- ➢ We implemented access tokens for the authentication. As mentioned in the User Account, the authentication step encrypts an unique user id as access token at the login or a successful registration, and decrypts received tokens to uuids to authenticate clients.

Fancy Graphical User Interface

- ➢ We enhanced the user interface to a fancy and pretty Google-drive-like file system and Gmail-like mail system graphical user interface.

Major Design Decisions and Major Challenges

- ➢ File transmission: File uploading and downloading involves binary transmission. When uploading binary files (PDF, images, etc.), the request HTTP header's *content-type* should be *application/octet-stream*. And when downloading binary files, the request HTTP header should contain a field *responseType: 'blob'* to ensure that the HTTP response can be resolved as a binary file instead of a garbled string.
- ➢ TCP connection management: To simplify and explicitly manage the tcp connection between each server, we self-defined two kinds of tcp connection: in and out.
  They are wrapped as InTcp and OutTcp classes separately (see tools.hpp). It is because the heartbeat will be sent periodically, and we do not want other message block the tcp used for heartbeat. Each server maintains a separate for both in and out tcp connections. When a server accepts a connection from other server, it creates a new `InTcp` pointer. When it needs to connect to other server, it creates a new `OutTcp` pointer. Generally, when a server no longer receives a heartbeat from another server after a timeout, it will delete the `InTcp` , and disconnect the `OutTcp` connection.

Team Members and Responsibilities

Bowen Jiang: HTTP servers for User Account / File / Folder / Mail

Wendi Zhang: Key-Value Store, SMTP

Zheng Han: UI and API design for User Account / File / Folder / Mail / Admin Console

Zhijun Zhuang: Admin server, load balancers, replication, log & checkpoint, heartbeats