CIS 505 Final Project Report (Team T15)

Xinyuan Zhao, Zhonghao Lian, Duy Duc Doan, Alex Hirsch

Feature Overview

In this project, we have implemented the PennCloud system providing email service and file storage for multiple users, supported by an underlying key-value database.

1. Account Service

Users can perform the following actions regarding their accounts:

- Register for a new account with username and password authentication
- Log in using the registered username and password
- Once logged in, users have the option to change their password

2. Email inbox

Once logged in, users can perform the following actions with their email account

- View all the emails they have received
- Send an email to another users in the same PennCloud system or a remote email address
- Delete multiple emails at once
- Extra Credit: Add more styling to the page to make it prettier

3. File storage

Once logged in, users can manage their files as below

- Create a new folder (under any hierarchy)
- Delete a folder
- Upload a new file, file can be text and any kind of binary files
- Delete a file
- Download a uploaded file
- Extra Credit: Each user is assigned a quota of 30MB, users cannot upload once this quota limit is exceeded
- Extra Credit: There is no restriction on the size of the file user can upload as long as he/she still has enough storage quota

4. HTTP Web Server

- Handle data transfer between browser and frontend server
- Load balancing to direct the load to nodes with the least amount of traffic
- Extra Credit: Persistent HTTP connection

5. Consistent and Fault Tolerance Database

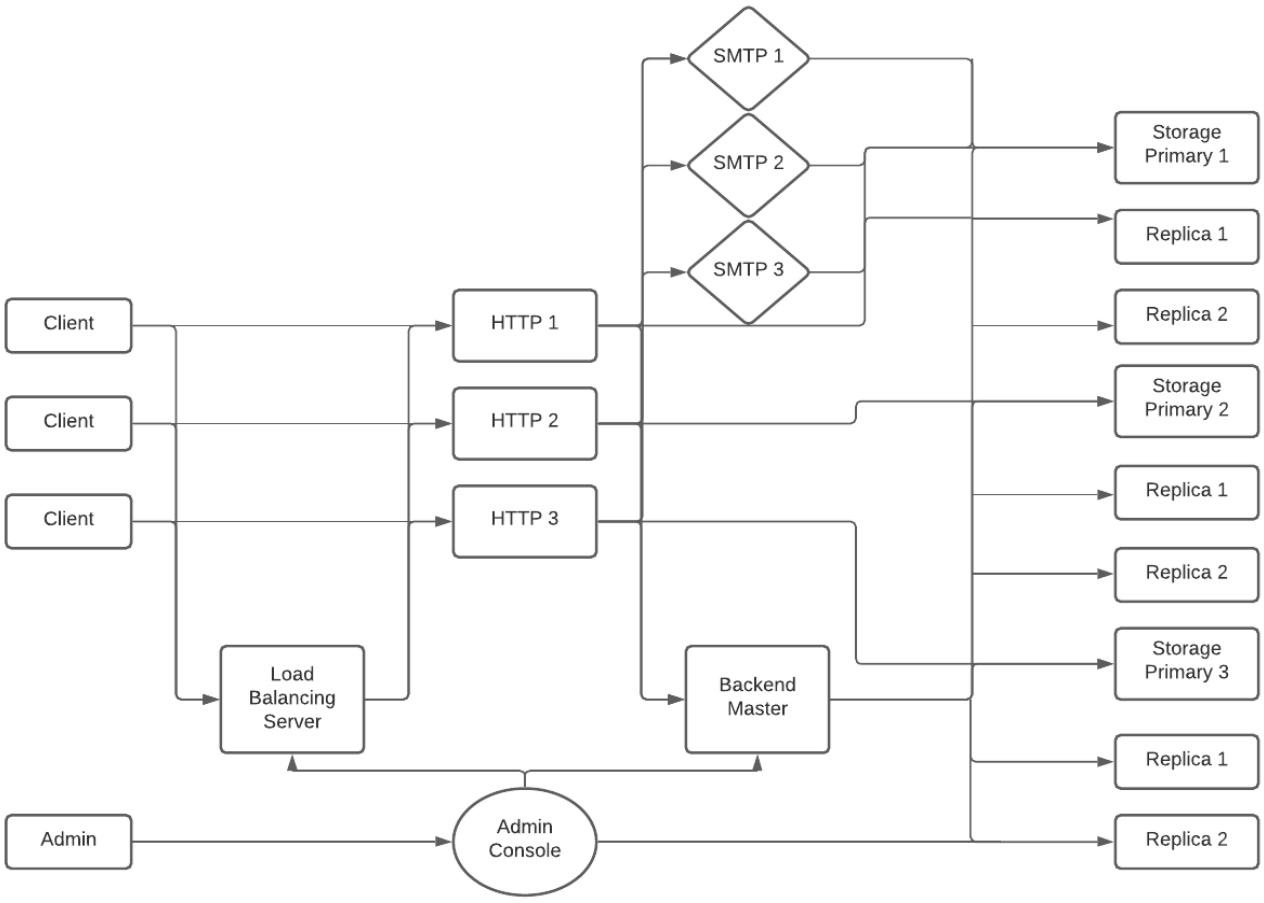
- A distributed key value store that can store any byte data
- The database is consistent and fault tolerant using the technique of logging, replication and checkpointing

6. Admin Console

The system also provides a console to manage the state of current system

- User can view the heartbeat status of nodes on both frontend and backend
- User is notified when any backend node is checkpointing or recovering
- User can Disable, Restart and View Raw Data for any of the backend nodes

Architecture Design and System Components



1. Frontend Web Server

1. Implementation
    a. A multithreaded TCP server handles incoming clients and distributes them evenly to the three frontend servers. Distribution is always to the frontend server that has lowest load

at last communication, measured as the frontend server with the fewest client threads currently running. Messages from the frontend nodes with load values are transmitted every second. If the load balancer does not hear from a frontend within three seconds (three missed messages), it classifies the frontend server as “down” and does not send any new clients to it, providing fault-tolerance. A daemon thread checks the time of receipt of the latest frontend message from each frontend node every second and updates the nodes classification accordingly. A heartbeat thread sends gRPC messages to the admin console to maintain active frontend node state.

   b. A multithreaded TCP server handles incoming client connections. Each incoming connection spawns a new worker thread and adds the thread to an unordered set of live threads. When the thread completes, it is added to a set of dead threads which a daemon thread periodically cleans up. This allows a heartbeat thread to communicate to the load balancer every second how many threads are currently active.
   c. The server supports HEAD, GET, and POST requests, and maintains persistent connections with clients for up to five seconds.

2. Challenges
   a. The biggest challenge was parsing HTTP messages and using them to populate HTTP responses. Fortunately, since messages are text based, it was relatively straightforward to debug. Managing live and dead threads, and cleaning up dead threads, was also tricky to get right, particularly because Firefox sends HTTP requests that are unprompted by the end-user. Finally, the algorithm for ensuring the load balancer was always ordered was tricky, but using a priority queue simplified the implementation; it also reduced efficiency due to the need to replace entries on every message, but this was not a large issue due to the small number (three) of nodes.

2. SMTP Server/Client

1. Implementation
   a. An TCP server is launched with each frontend node and waits for incoming client connections. When connections come, it communicates via the SMTP protocol and saves emails to valid users’ email boxes.
   b. The frontend server also includes SMTP client functionality that looks up DNS records for non-local email addresses, and supports sending, replying (all), and forwarding among valid PennCloud user accounts. Users can also delete multiple emails at once.
2. Challenges
   a. Fortunately, a team member (Xinyuan), had a superb SMTP implementation which was readily integrated with our frontend leading to no real challenges in this aspect of the project.
   b. Looking up MX and DNS records was difficult, but the sources cited on Piazza and other students’ comments were extremely helpful. Figuring out the right combination of behaviors to get external email servers to accept messages was also tricky.

3. Storage Service (implement in Frontend server):

1. When HTTP server receives a POST request that has content-type: multipart/form-data in it, it will pass the request body (with file binaries) to storage service; the storage service then

parses file content according to the multipart boundary, store the data to backend. When the file size is large, the backend will automatically store the file content in several cells.
2. Every file stored in DB has a parent id field in its metadata, denoting in which folder the file exists. When user opens the files service, the storage service will first fetch all the file metadata that belongs to this user, and then construct a multi-layer folder structure with file id and its parent id
3. When user moves file around, we simply change its location in folders by editing the parent id field in its metadata
4. Everytime the file service is opened, the service will calculate the total size of files uploaded. When it exceeds the assigned amount, the upload function will be diabled

4. User Account Service (implement in Frontend server):

1. Implementation
    a. When users register accounts, the frontend checks a special column in the kvstore store to see if the username already exists. If it does exist, the user is prompted with a pop-up notification that the username is already taken. If not, it stores the username and password. When a user is logged in, the frontend assigns the user a session ID cookie and marks that session ID as logged in. Using cookies, each frontend node can maintain multiple logged in users simultaneously. When users log out, their session ID is removed from the “logged in” map. Users can change their password, which is implemented with CPUT.
2. Challenges
    a. Thanks to hash functions written in previous homeworks, creating session IDs was trivial. Maintaining sessions was also straightforward. Parsing HTTP requests with cookies was the most tricky aspect of implementing this feature.

5. Web Server and Database Communication

1. A database client interface is implemented as part of the frontend to communicate with the database backend. It provides the following functionalities:
    a. Initialization: The client can be initialized with the address of the master node which is used to establish the connection to the master server. When there is a data request, the client will ask the master which is the node that the request should be sent to and then cached this information so that the master node is not involved in all the requests. In case the cached data node is down, the client can detect and ask the master node again for a new data node to send to.
    b. For general string: GET/PUT/CPUT/DELETE requests for any string data
    c. For email data: users can create/delete/list all emails. The email data is serialized/deserialized to/from string before saved in data nodes.
    d. For file data: We split the file's information into file metadata (which contain information such as title, size, etc.) and actually file content. The communication between frontend and master node as well as data node is wrapped into this client interface using gRPC messages and services.
2. Challenges:
    a. Implementation of large file upload/download: since the file is too big and go over the limit of allowed message size of gRPC, we need to split the file content into multiple

cells and stored as different parts in the database. This would put no hard limit on the size of the file that can be supported by the system. Each chunk will be attached an id, so that they can be resembled during retrieval.

6. Database Server

Key-Value Store process in single node

1. In each database node, data will be stored in memory: new values sent to the database will be kept in a local map in each of the node servers where each cell is characterized by row and column keys. Data is partitioned into groups of data nodes based on the first character of the row key.
2. When there is a data request (GET/PUT/CPUT/DELETE), the map will be queried/updated accordingly. Each update request will be recorded in the log file with a system-tracked sequence number
3. Checkpoints will also be taken based on the amount of new data accumulated. These checkpoints will be used in recovery. The checkpoint is done in 2 phases: first the primary will initialize the checkpoint by sending a request to the replicas, then once all confirmed, the primary will issue the real checkpoint request and wait for all to finish. Checkpoint is a flush of data currently stored in the map.

Multi-threaded Master Node and Load Balancing:

1. For the backend, a single master node will be responsible for redirecting the client's request to a primary storage node according to the first character of the username.
2. We assume that the master node will not fail. And to reduce the risk of bottleneck, the master node will only handle the request when a user logins in or registers for a new account. The request directly goes to the storage node if it has the same session ID.
3. The Master node is also responsible for checking the status of each storage node periodically. For every 6 seconds, it communicates using gRPC with each storage node to retrieve their heartbeat, checkpointing, and recovery status. These status info can be accessed through Admin Console.

Replication

Primary Based Replication: We will have a flat layer of Storage nodes.

1. There are 3 primary nodes and each of them have 2 replicas.
2. Primary nodes will handle requests from the front-end and send synchronous updates to its replicas. This will be performed in 2 phases: the primary node first send the init request to its replicas to acquire the necessary resource (mutex on the current executing request sequence number), then it will forward the update requests to the replicas then wait for confirmation from every replica

Consistency

Sequential Consistency: Use mutex on rows to make sure concurrent writes to each row will be consistent. In addition, each update request is attached with a sequence number which will also be recorded in the log file, which will be used in recovery.

Fault Tolerance and Recovery

1. When a primary node is down, another running data node will be selected as the next primary node. Since all the write requests are replicated, the data will be in sync.
2. When a node first starts up or restarts, it will ask the current primary whether its last known sequence number is the most updated. If not, the primary will send over the latest checkpoint and log over to the requesting node for recovery. The storage will be rebuilt from the checkpoint and replaying all the logged write commands (PUT/CPUT/DELETE).

Challenges

1. During recovery, a large amount of data needs to be transferred from primary node to replica node. In addition, there is no limit we should enforce to restrict the size of the log or the checkpoint
   Solution: Use streaming to split the recovery payload into multiple chunks to be sent to other nodes
2. Coordinate different states of the nodes: killed, recovering, committing, forwarding requests to replicas, checkpointing or awaiting more requests from primary (either checkpoint or commit). Communication between data nodes is implemented with gRPC.

7. Admin Console

1. Admin Console is using a special address (localhost:7070) and its own HTTP server to monitor the status of backend and frontend nodes. To retrieve the status information, the admin console will contact the frontend load balancer and backend master node when the user opens or refreshes the page.
2. The admin console also supports disable/restart and view raw data for each backend storage node. For disable/restart, the admin console sends a message to the node who will switch to sleep/normal mode accordingly. To view raw data, the backend node sends the lists of row key, column key and binary raw value to the admin console, and the admin console view raw page will display the data in tables and paginated by 10 items per page.

Challenges

1. It is hard to restart a node remotely, so we decided that when node receives a disable signal, it goes into a sleeping mode which does not handle requests and only listens to a restart signal.
2. For viewing raw data, large binary data is hard to render and display on the UI. And so we decided that we only display short data which contains 100 characters or less, otherwise we will just display that the file is too large.

Collaborations

- Alex Hirsch: Frontend Server Listener and Daemon, Email Server/Client, Load Balancer, HTTP Parsing, Frontend Worker Logic (with Zhonghao)
- Zhonghao Lian: Frontend Worker Logic, Views (rendering functions), Storage Service
- Duy Duc Doan: Database client, Key-Value Store, Logging, Replication, Checkpointing and Recovery, Large File Handle
- Xinyuan Zhao: Master node management and Admin Console

Note on recovery issue during presentation

Summary: When there are two nodes down, the first cluster is not recovered properly while the third cluster performs normally.
Our explanation for this can be the combination of following reasons:

1. Cluster 1 at that time is storing a large file (after testing uploading a ~30mb file), hence the replication/recovery/checkpointing will take longer time
2. However, we might kill the node(s) too fast while it has not finished replication/recovery leading to corruption of sequence number/log/checkpoint combination. Hence, even when we restart all the nodes again, it might still not recover again properly.