PennCloud- CIS 505 Software Systems Final Project Report

Spring 2024

Manas P. Shankar, Paul Kathmann, Priya DCosta, Rupkatha Hira

School of Engineering and Applied Sciences, University of Pennsylvania

Abstract :

The PennCloud system is designed to provide a robust and scalable cloud service, utilizing a distributed design for enhanced performance and isolation. Key components include Front-end Servers, Front-end Load Balancer, Backend KV Stores, Backend Master/Load Balancer, Email Servers, and an Admin Console, all interconnected using gRPC for efficient, language-agnostic communication.

Section 1 describes a brief overview of the design of PennCloud along with its various components and how they interact. Section 2 describes the key-features developed and implemented in the cloud service. Section 3 goes into the detailed description of the design components, underlining the key role of the individual components, along with some design choices and challenges. Section 4 touches upon the choice of communication protocol used, and Section 5 lists the contribution of each of the team-members in the development of the project.

1 Design Overview/Architecture

The project consists of two major parts:

(1) Frontend : Frontend servers that accept HTTP requests from clients; load is distributed amongst these servers through the frontend load balancer.

(2) Backend: Backend email servers for executing mail transactions, KV stores for data storage and a backend master for backend load balancing.

A client interacts with the system through the user interface, which parses HTTP requests and triggers commands to the back-end servers. The back-end server functions as a repository for all client data, utilizing a KV store system similar to BigTable. This data is distributed across three different servers, with a backend master responsible for managing the data distribution information.

The Front-end Layer comprises a load balancer and four front-end servers handling client interactions and distributing requests based on server load, improving responsiveness and availability. It serves as the user interface for account management, file storage, and email services.

The Backend Layer is central to data management, and has several components. It features the Backend Master which doubles as a load balancer, directing requests to appropriate KV Stores based on load and data locality. It manages node health and status, and orchestrates primary node elections among KV Store replicas to ensure data consistency and availability. Data is sharded and replicated across multiple KV Store nodes, which manage all data storage and retrieval operations, ensuring redundancy and high availability. We have demonstrated this using two replication groups each having three KV nodes.

The email servers(SMTP and POP3) act as the backend between the frontend email interface and the KV storage, that actually implement the protocols.

The Admin Console provides a graphical interface for system monitoring and management, allowing administrators to oversee system health and manage service components in real-time.

Overall, PennCloud leverages modern architectural principles to deliver secure, resilient, and scalable cloud services, ensuring efficient operation and seamless user experience. We will be elaborating on the design of individual components in the following sections.

2 Key Features

The key features which our cloud service provides are as follows:

- **User Signup/Login** Provides the interface for new users to signup and existing users to login. Login authentication is implemented here.
- **Drive Storage**
  - A web storage service that is a Google Drive mini-clone, with functionality for uploading, downloading, moving, deleting and renaming files as well as folders.
  - Our implementation supports nested folder structures, files well above 50 MB in both binary and text formats, as well as the ability to rename nested files and folders.
- **Front-end Load Balancer** Balances the load and appropriately routes the requests from clients that connect to the system to any of the 4 frontend servers. It directs the incoming requests to the Front End Server + Controller.
- **Front-end Server + Controller** This node contains all the https endpoints that the client can call (e.g. GET /login, GET /home, POST /data…), as well as the controller logic of how to handle these requests.
- **Backend-end Master/Load Balancer** Acts as a load balancer for the KV stores, and provides an interface to access the status of the KV nodes from the frontend/admin console.
- **Key-Value Store** Stores and returns data in key-value format similar to Google Bigtable(Chang et al., 2008).

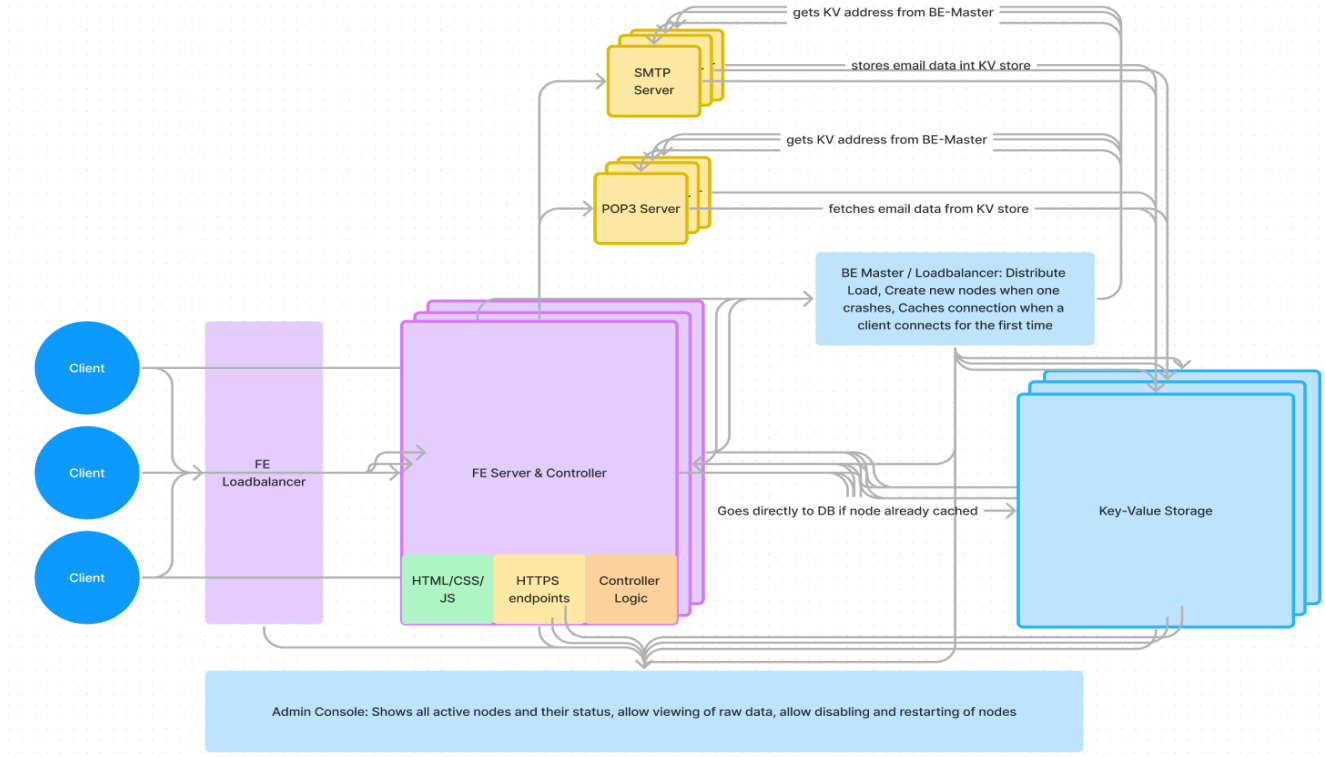

Figure 1. Architecture Diagram.

  - Endpoints for reading and writing data
  - High Availability and fault tolerance
- **Email** A web-mail service allowing users to send, receive, view and delete emails from other users.
- **Admin Console**
  - A dashboard allowing uses to view the status of different nodes, shutdown and restart a node.
  - Also allows for viewing of the raw data within the KV storage.

3 Design Description

This section describes the detailed design of the key components of our system.

3.1 Front-end Server

Frontend Login:
The Frontend login and registration functionality is based on PUT and GET calls to the KV-Store's gRPC server. For login, we first check if the user exists using a GET call. If they do not exist, we redirect the user to the registration page, where a PUT call is used to add the user and the password to their account in the KV-Store.

However, if the user tries registering with pre existing credentials, we confirm with a GET call that the user does indeed exist, and accordingly direct the user to the login page. In all other cases, we register the user.

Front-end File Storage

The Frontend webstorage service presents a drive-like interface to the user, with the option of downloading files from the KV-Store, uploading to the KV-Store, creating folders, renaming files and folders, moving files and folders to within directories (creating a nested structure), and deleting files as well as folders.

To enable efficient access and to simplify interactions with the KV-Store, we decided to use a structure reminiscent of a mapping structure between the file/folder names and a unique hash ID. This map would in turn in serialized and stored as a string within a cell in the KV-Store identified by the (username, hashID of the directory). The contents of any given file would also be stored using a similar indexing terminology.

All operations involving moving, renaming and deleting first require a search functionality to be set up. In this instance, we make use of a graph-like interpretation of the file directory, particularly a tree, since we are dealing with a directed acylcic graph. Where the root node is connected to different files and folders with their respective hash IDs. The file nodes are essentially leaf level nodes since they hold the file content, whereas the folder nodes are intermediate levels of the tree, with further levels of file and folder nodes. The search operation involves performing a BFS-traversal of the nodes beginning from the root directory. Files are located and returned immediately, whereas maps associated with nested folders are added back to the queue for processing in the next round. We note that this implementation removes complexity associated with having to rely on nested hierarchies within the KV-Store and the frontend server, with the tradeoff of slightly increased complexity in lookups.

For move operations, 3 searches are performed, to find the file, source folder and the destination folder. Then, the name and

hashID of the file is simply appended to the destination folder's map structure and the string is re-serialized. For renaming, a single search operation is performed to find the file/folder location, and the key value in the map entry is replaced with the newer file/folder name, with the hashID remaining the same. Thus, the identification of a file/folder is constant throughout the system, making it persistent in the face of constant renaming and from duplicately named files/folders. Finally, the deletion operation involves again finding the file/folder. For a file, the DELETE operation is simply performed. For a folder, all nested files and folders are first recursively deleted, and only then is the top level directory purged.

It is worth highlighting here that our design philosophy, along with the use of gRPC for communication and file transfers has allowed us to considerably cross the 10MB stated threshold. Currently, incoming message sizes have been set to 1GB on both the client and the server, and in practice, we found that we were able to regularly upload files well over 120MB in about 15-20 seconds.

We also note that there is considerable room for improvement, particularly with respect to the time complexity of carrying out nested searches for files/folders, as well as improving the throughput associated wth transferring large files.

Design Decisions and Challenges:

As described in the design section above, we decided to make use of a serialized mapping structure to store file/folder names with their associated hashIDs. While this eliminated the need for complex hierarchies on the frontend server and the KV-Store, it did introduce the added complexity involved in making BFS traversals across the file structure in order to locate and perform operations on any file/folder.

The performance benefits involved in using gRPC, as well as the associated simplifications in creating services cannot be understated. However, there was the added challenge of ensuring consistency in the respective proto files used across all components. Additionally, there was also the challenge of maintaining consistent gRPC installations across machines, in order to be able to compile and execute each other's code.

In terms of actual file transfers, we noted above that we enabled support for incoming messages of upto 1 GB. However, we found that beyond 200 MB, latency considerably increased, thus impacting user experience. Another design challenge was to ensure efficient transfer of files, which we solved by using byte encoding of the message contents between services. However, an improvement would necessitate making use of bytestreams or even chunking in order to improve transfer speeds while maintaining integrity.

3.2 Back-end Master

Backend Abstraction:
The Backend Master distributes the load amongst the backend KVs and keeps track of all the nodes across different replication groups. It hides the implementation details like replication, recovery, etc from the frontend, and simply gives the frontend server access to a KV node that it can then use to successively query for useful data, performing operations like GET, PUT, etc.

HeartBeat Monitoring and Primary Allocation:
The backend master accesses config files for each group, to get information about the KV nodes in that group. It keeps track of all the active nodes in each group by sending heartbeat requests to each node periodically. It also assigns one of the nodes in each group as the "primary", and sends the information about the primary as well as all the active nodes to each node of the group, so that the nodes can communicate with each other and the primary, for replication and recovery purposes. The "primary" is chosen randomly among the "active" nodes in a group. It is reassigned if the current primary crashes.

Backend Load Balancing:
The frontend server queries the backend master using the row-key of the data it wants to access. The backend master keeps a map of row-key ranges to a particular group of KV nodes. Whenever the frontend server queries the backend master with a row-key, the backend master first gets the group that would handle the row-key, and then returns the address of a random node among the active nodes in that particular group.

Thus, the backend master offers load balancing at two levels:

- The row-key ranges are split up across groups, hence different KV groups can handle different sets of row-keys.
- Within a group, the backend master returns one of the addresses of active nodes at random. Hence the load among the KV nodes in a single group is uniformly distributed.

(N.B.- It acts the same way for the email server, returning the address for a particular row-key(username) that the email server queries)

The backend master acts as the gRPC server for the Frontend server, the Email server, and the Admin Console. For the first two, it returns the address corresponding to a particular row-key, while for the Admin Console it returns the address of all active nodes corresponding to each group.

It acts as a client for the KV stores, requesting health checks periodically and preemptively sending the active nodes and primary node, in case any node crashes.

Design Decisions and Challenges One of the design decisions we made regarding the interaction between Backend Master and KV nodes, was to delegate the tracking of active KV nodes and assigning of the primary node to the Backend Master. The Backend Master would then update all replicas in the group whenever there is a change to the active nodes or primary node, so that the replicas only communicate with active nodes and don't block waiting for the response of inactive nodes.

We tackled the backend load-balancing within a replication group by randomly picking a KV node within a group to handle request from the front-end, instead of some more load-sensitive approach like choosing the node with the least amount of load.

3.3 Key-Value Storage

Data Storage and Tablet Design

The KV acts as a database storing all the user data and files. We distribute the data across 2 groups of 3 KV replica nodes each. Each group will be assigned a key range they will store e.g. "a"-"m". All requests with keys in that range will go to one of the

3 replicas of the group (Backend-Master decides which replica the request would go to).

Each KV node stores the data in tablets with each tablet containing up to 200MBs of data. Each tablet stores the data in rows of key-value pairs, where the keys are strings (usually the username string) and the values are lists/vectors of more key-value pairs (each key represents a column). The values of the inner key-value pairs can contain any kind of data stored in byte format. The key-value pairs of each tablet get encoded in byte format and stored in string-sorted order on disk. Whenever we need to access the data to read or write from it we load the data into memory and then read or apply the changes. We keep the tablet in memory until data from another tablet on that node is accessed, to minimise disk operations and speed up processing.

We also ensure that we never lose data from any tablet by always logging all write operations in a log file for each tablet and replicating write operations across all 3 replicas. Thus, even when one node dies the system can process any write/read operations using the latest data. The system is designed with scalability in mind: Whenever a tablet hits the maximum storage size of 200MB it will initiate a partition of the tablet into two tablets. A new tablet will be created and the second half of the data will be transferred to the new tablet and removed from the old tablet. A Tablet Manager Singleton class which keeps track of all the tablets will also update the key range that each of the tablets contain so that we can easily find the data after partitioning and don't have to search all the tablets.

Fault Tolerance

We designed the system to be fault-tolerant for cases where one or several of our KV nodes crash. We have achieved this by implementing replication and recovery protocols as well as adding centralized checkpointing and logging mechanisms into our KV nodes. Thus, we ensure data written to our KV never gets lost and even when 2 out of 3 replica nodes crash we always have the latest data available for the users.

Replication

To ensure all the active replicas always have the latest data and are in sync with each other, we implemented a primary-based replication protocol. Whenever the secondary replicas receive a write request they will first forward the request to the primary replica (the master node decides who is the primary and informs the other nodes). The primary replica will then multicast the message to all replicas, which in turn will write the message to the respective tablet and then return true to the primary replica. Once the primary replica receives success messages from all secondary replicas it returns true to the original forwarding node so that node can finally return a success message to the client after the data has been written to all replicas.

Checkpointing and Logging

Since the nodes use in-memory processing of operations we had to ensure that no data gets lost from memory if a node crashes. We have achieved this by checkpointing all write operations to a log file that can later be used to recover any transactions that had been applied to memory but weren't checkpointed/written to disk before the node crashed. We store a log file for each tablet to simplify the recovery process.

Additionally, we regularly checkpoint the log files and in-memory tablets to disk. Instead of choosing a time-interval-based checkpointing we choose to track the number of write transactions since the last checkpoint and then checkpoint after hitting a given limit of c write transactions, where c is a tuneable parameter given as an input to the KV program from the command line. This design choice would save resources when there are few writes to the system and ensures safety when there are a high number of write transactions (by checkpointing more often when write frequency is high).

The checkpointing is initiated by the primary node whenever we hit c write transactions and is implemented following a centralized checkpointing protocol. The primary node will multi-cast a checkpointing message to all the replicas which will then start the checkpointing and return true once checkpointing has completed. We only allow read operations during the checkpointing operation to ensure data consistency.

Recovery

Whenever a node dies and the remaining nodes receive more write messages until the crashed node restarts, we need to ensure the restarted node recovers all the missing data from the active nodes. We achieve this by comparing the checkpointing number of the recovering node with the one of the primary node (the backend master ensures only active and recovered nodes can be the primary). If the checkpoint number of the primary node is the same, then we know the checkpoint files are still in sync and we only need to forward the newly added log file entries to the recovering node. If the checkpointing number of the recovering node is behind the primary node, then the checkpoint file is also out of sync and we need to send the checkpoint file as well as the log entries to the recovering node. Once the recovering node receives these it will write the received log files and checkpoint files to disk. Once it has done this we can mark the node as recovered and active with the backend-master which will inform all other nodes.

Design Decisions and Challenges Due to time constraint we decided to follow a primary-based approach for checkpointing, replication and recovery which was faster to implement, but adds extra load on the primary replica potentially slowing down the system if they primary has to do a lot of work. However, this decision allowed us to build a fault tolerant system within a few weeks. Given more time we would consider switching to a 2PC commit protocol for replication for cases where a transaction could go through on one replica but not on the other for some reason.

To simulate the shutting down and restarting of the KV nodes without actually physically crashing the system, we introduced daemon servers that ran shutdown/restart services for each of the nodes.

Using gRPC saved us a lot of time (no need to serialize objects etc.), but also came with new challenges like ensuring all programs (frontend/backend/admin-console) have access to the most up-to-date versions. We also had issues getting the chunked file streaming to work for our recovery protocol. Thus, we had to simulate streaming by looping over all chunks and calling the gRPC stub for each chunk.

3.4 Admin Console

The admin console receives the status of all the backend KV nodes from the backend master, and the raw data in the KV from the KV using gRPC calls. Further, it uses gRPC calls to communicate when the shutdown or restart button is hit by the user.The admin console page is refreshed every time a button is clicked or the page is refreshed, fetching the latest data from the KV and the latest node statuses

3.5 Email

Front-end Email servers use the SMTP and POP3 protocol to allow users to send and retrieve emails. The frontend email server sends the appropriate commands to the backend email server depending on the users' action. For example, when the user reaches the inbox page, the frontend email server will send USER, PASS, STAT, LIST commands to the backend email server to retrieve and display emails for that user. The front-end HTTP servers fetch the stored emails from the key-value store to the user. The client only interacts with the front-end HTTP server for all email transactions.

- **Inbox**
  - **POP3**
    - **USER and PASS** - To login for a particular user.
    - **STAT** - To get the number of emails for that user.
    - **RETR** - To retrieve the contents of each email, one by one.
    - **QUIT** - To quit the email server (since this is a stateless model).
- **Compose/Send Email**
  - **SMTP**
    - **HELO** - To establish a connection with the SMTP backend.
    - **MAIL FROM** - To send the sender of the email.
    - **RCPT TO** - To send the recipient(s) of the email.
    - **DATA** - To send the actual content of the email.
    - **QUIT** - To quit the email server (since this is a stateless model).
- **View Email**
  - **POP3**
    - **USER and PASS** - To login for a particular user.
    - **RETR** - To retrieve the contents of the specified email.
    - **QUIT** - To quit the email server (since this is a stateless model).
- **Reply to Email**
  - **POP3**
    - **USER and PASS** - To login for a particular user.
    - **RETR** - To retrieve the contents of the specified email.
    - **QUIT** - To quit the email server (since this is a stateless model).
  - **SMTP**
    - **HELO** - To establish a connection with the SMTP backend.
    - **MAIL FROM** - Current user's email address.
    - **RCPT TO** - Using the FROM address from the retrieved email.
    - **DATA** - To send the actual content of the email.
    - **QUIT** - To quit the email server (since this is a stateless model).
- **Forward Email**
  - **POP3**
    - **USER and PASS** - To login for a particular user.
    - **RETR** - To retrieve the contents of the specified email.
    - **QUIT** - To quit the email server (since this is a stateless model).
  - **SMTP**
    - **HELO** - To establish a connection with the SMTP backend.
    - **MAIL FROM** - Current user's email address.
    - **RCPT TO** - Email address provided by the user.
    - **DATA** - To send the actual content of the email.
    - **QUIT** - To quit the email server (since this is a stateless model).
- **Delete Email**
  - **POP3**
    - **USER and PASS** - To login for a particular user.
    - **DELE** - To delete the current email, using the relevant email number.
    - **QUIT** - To quit the email server (since this is a stateless model).

Back-end The backend email server performs actions depending upon the commands received from the frontend email server. For example, the SMTP server stores an email to the KV store upon receiving the DATA command. Similarly, the POP3 server retrieves data from the KV store upon receiving the RETR command.

The SMTP server is built in compliance with RFC 821[*Simple Mail Transfer Protocol* 1982], and POP3 server is built in compliance with RFC 1939[Myers et al., 1996].

For storing or accessing the email data, the email server queries the backend-master with the username, to get the address of a KV store to contact. It then stores at/retrieves from that KV store, the corresponding data, using PUT/GET/DELETE etc.

Recovery Even if the KV node the backend server contacts, crashes, the frontend retries and the backend-master returns the address of one of the active KV nodes to the backend server, which is then used to store/retrieve data. This is made possible by the replication/recovery feature implemented at the KV store.

Design Decisions and Challenges For view, reply, forward and delete emails, the relevant email number was included in the HTTP request using the HTML form tag, making it easier to retrieve that particular email through POP3 commands.

4 Cross-Server communication

Due to the high number of messages being sent for communication between nodes and for the implementation of the different protocols we decided to use gRPC for Cross-Server communication. Thus, we didn't have to spend any time serializing objects that we wanted to sent between the nodes. As gRPC is also multithreaded by default and uses threadpools to manage requests we saved some time building out the boiler plate code for multithreaded message handling. However, abstracting this away also made debugging more challenging, whenever there were bugs related to multithreading.

5 Contributions

All team members contributed equally as follows:

- **Manas Shankar (Frontend):**
  - Frontend user registration, login, HTTP request parsing and controller interface definition, including all related HTML and CSS files (used Bootstrap for styling).
  - Webstorage service, including all related HTML and CSS files (uses Bootstrap for styling).
- **Priya DCosta (Frontend):**
  - Frontend email server, including related HTML and CSS.
  - Admin console and related HTML and CSS.
- **Paul Kathmann (Backend):**
  - KV Storage, including replication and recovery.
- **Rupkatha Hira (Backend):**
  - Backend master.
  - Backend email server.
  - Shut-Down/Restart functionality in KV store.

References

Chang, Fay et al. (June 2008). "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: https://doi.org/10.1145/1365815.1365816.

Myers, J. and M. Rose (1996). *RFC1939: Post Office Protocol - Version 3*. USA.

Simple Mail Transfer Protocol (Aug. 1982). RFC 821. DOI: 10.17487/RFC0821. URL: https://www.rfc-editor.org/info/rfc821.