Technical Report: Design Brief

Rishi Ghia , Pranshu Kumar , Zihao Deng, Janavi Chadha

May 14, 2024

1 Introduction

For this project, our team worked to create a smaller version of Google's cloud services specifically implementing Gmail and Google Drive. Our platform, PennCloud, includes a mail service where our users can send emails both to users within PennCloud and a storage service. Below is a diagram showing how our system works. When a user logs into our system they are redirected by way of the load balancer to one of our frontend servers. From the frontend server they are able to either send or receive emails by way of the SMTP server, or upload documents. These requests are sent to the backend server as outlined in the diagram. The requests are specifically sent to the master node which then based on the user's username will connect them to a primary server within a replica group.

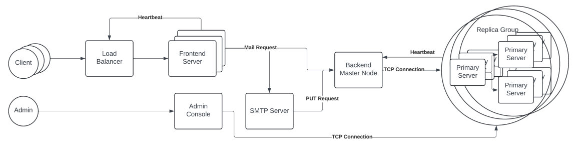

Figure 1: PennCloud Diagram

2 Major Components

2.1 Key-Value Store

2.1.1 Overall Structure

The backend consists of several multithreaded, distributed servers. There is one master node, which is connected to several replica storage nodes. There are n replica groups in total, with each containing one Primary server, and several secondary servers. These servers, groups, and their respective IP Addresses, TCP and UDP ports are all defined inside a `config.txt` file.

2.1.2 Backend Master Node

The Backend Master Node has several different features and responsibilities:

- **Load Balancing:** The frontend server sends a request over TCP to the master node, along with a username. The master node, based on the first letter of the username (load balancing), sends back a replica server's TCP and IP, which is chosen using a round-robin method in order to minimize the maximum load on any server.
- **Primary/Secondary Assignment:** The master node is also responsible for storage node status assignment to either Primary or Secondary. The first connecting server from any replica group is assigned as the primary, and others as the secondary.
- **Dead Server Recognition:** If a server's heartbeat is not received for more than 300ms, then it is marked as dead, and then the master reassigns the primary to the next secondary server that is connected to the master.
- **Providing Admin Console Server Details:** The master node also received the request from the Admin Console for the `config.txt` file, which then sends it.

2.1.3 Replica Nodes

The key responsibilities for the replica nodes are:

- **Data Storage and Recovery**
  - Maintains an independent copy of the data in a disk file.
  - Employs either checkpointing or a write-ahead log (WAL) for efficient recovery in case of failures.
- **Write Operations**
  - Prioritizes in-memory operations for performance.
  - Batches multiple write operations together before persisting them to disk, optimizing disk usage.
  - Handles the persistence process asynchronously to minimize impact on request handling.
- **Command Handling**
  - Implements handlers for GET, PUT, CPUT, DELETE, and LIST commands over TCP connections.
  - Includes robust error handling for invalid commands or operations.
- **Replication and Consistency**
  - **If designated as primary:**
    - Initiates 2-phase commit for write operations, ensuring coordination with all secondaries.
    - Commits changes locally only after successful coordination.
    - Sends the operation outcome to the client/frontend.
  - **If designated as secondary:**
    - Participates in the 2-phase commit process initiated by the primary.
    - Persists changes to its local store based on the primary's commit/abort instructions.

2.1.4 Communication within the Backend

There are 4 different means of communication implemented within the backend:

- **TCP Port 1:** Used to send and receive commands and data from the Frontend servers, such as PUT, GET, DELETE, CPUT, LIST and GETALL. A buffer size of 16384 was used along with chunking of the file while sending and receiving, to ensure maximum transfer speed.

- **TCP Port 2:** Used to send and receive ENABLE and DISABLE requests from the admin console. A disable request essentially renders the server dead, while still keeping it alive. This is done by busy waiting on each of the communication ports, ensuring that no requests can be received, and no heartbeat can be sent.

- **UDP Port 1:** This port is used by the storage nodes to send heartbeats to the master node at an interval of 100ms each. If no heartbeat is received by the master node on this port for over 300ms, the storage node is marked as dead.

- **UDP Port 2:** This port is used by the master node to send communication to the storage nodes, indicating the status they're supposed to take on (such as primary or secondary), as well as information about any server that might have died, or the new primary server that has been assigned.

2.1.5 Replication

Replication in the backend is performed using the technique of 2PC (2 Phase Commit). Whenever a write command is received by any storage node, the following can happen:

- **If the receiving server is a Primary Server:** It sends the information over to all secondary servers that it has received a command. Once all the secondary servers reply back with a "+OK", and only if they do, the data is written to all the servers within that replica group.

- **If the receiving server is a Secondary Server:** It sends the information to the primary server, and then the entire above point is followed, and 2PC is done. The secondary doesn't immediately just write the data.

2.1.6 Checkpointing

- **Operation Tracking:** Each replica server maintains a count of write-style commands executed in-memory. These commands include PUT, CPUT, and DELETE operations which modify the data.

- **Threshold-Based Persistence:** When the count of executed write commands exceeds a predefined threshold (in our case, 5), the replica server triggers a checkpointing process. This threshold balances performance with the frequency of persisting data to disk.

- **Data Dump:** During a checkpoint, the server's entire in-memory data representation (likely a tablet-like structure) is written to its disk file. This ensures that even in the event of a failure, recent changes are not lost.

- **Log Flushing:** After the in-memory data is saved to disk, checkpointing logs are flushed. These logs likely track the operations performed since the last checkpoint, helping with recovery if needed.

2.1.7 Crash Recovery

- **Initial Contact:** Upon restarting, the replica server contacts the current primary server of its group to inquire if any write-style operations have occurred since its last crash.

- **Synchronization (if needed):**

  - If the primary indicates updates, the recovering server requests a copy of the primary's current disk file and associated checkpointing/log files.
  - The server processes the received disk file to reconstruct the base data and then replays operations from the logs to synchronize its state with the other servers.

- **Local Recovery (if no updates):**

  - If the primary indicates no data changes since the server's last offline state, the recovering server proceeds to use its own locally persisted data.
  - It first loads its disk file and then replays its local checkpointing logs to reach an up-to-date state.

2.2 Frontend Server & Load Balancer

2.2.1 Load Balancer

- **Coordinator:** The load balancer routes the incoming user to one of the active frontend servers and maintains a reasonable workload balance among all the servers. Our design for achieving this is to have a special coordinator server that will accept all initial connection requests from users and route them to the appropriate frontend server, so that all subsequent requests from users go directly to the assigned server.

- **Communication:** In particular, the coordinator listens on a TCP socket and accepts all initial requests from the clients, and it uses another UDP socket to communicate with all frontend HTTP servers. To monitor and split the workload among servers, the coordinator communicates with each server to get the current number of users and their status.

- **Balancing Algorithm:** In the case where the workload of one server is much heavier than that of the others, the coordinator will not assign new requests to that server. Instead, the new users are routed to the server with the lightest workload, so we can ensure all servers get even workloads and no single server is significantly overloaded.

2.2.2 Frontend Server

- **HTTP Interface:** The frontend HTTP server is responsible for handling HTTP requests from clients and serving the web pages for users to interact with their drive and mailbox using a TCP socket. All functionalities including page navigation, form submission, and file upload are implemented by parsing the corresponding GET or POST requests sent by the client, and then responding with information retrieved from the backend. Regular text inputs from the clients are simply extracted from the HTTP request body, and files will be uploaded as multipart/form-data, where the request body contains a special boundary string to separate the file descriptions from the actual binary file data.

- **Load balancer/Admin communication:** In addition to interacting with the clients, the frontend server also uses a UDP socket to communicate with the load balancer and admin console. If the frontend server receives a heartbeat message, it then sends back the number of clients currently connected to it to the load balancer. If it receives an ENABLE or DISABLE message from the admin console, it then restarts or shuts down itself.

2.3 User Accounts

PennCloud allows multiple user accounts, ensuring personalized access to the webmail and storage services. When a user first connects to the frontend server, the server responds with a web page containing input fields for a username and password. After submission, the server verifies the credentials against stored values in the backend key-value store using the GET command.

If the authentication is successful, the server provides the user with a menu offering links to their inbox and file folders. The frontend server maintains user sessions to keep track of active users and ensure a smooth user experience. If the username is incorrectly entered, the user gets an error message and will be routed back to the login page. The system stores passwords along with most data in base64 values, ensuring that user data remains protected.

Users can sign up for new accounts through a similar interface, where their details are added to the backend store using PUT commands. Existing users can change their passwords by updating their stored credentials, which involves a sequence of GET, verification, and PUT operations to ensure consistency.

2.4 Webmail Server

PennCloud's webmail system provides users with email functionality. Users can view their inbox, send emails to other users within the platform, and communicate externally via email addresses outside the system. The webmail interface displays a structured list of email headers, showing sender, subject, and arrival time. The users can click on messages to view their content and perform actions such as delete, reply, or forward.

Emails are sent using SMTP (Simple Mail Transfer Protocol). When a user sends an email, the frontend client constructs the email using the SMTP protocol and sends it to the SMTP server. The SMTP server acts as a relay, connecting to the backend master node to determine the appropriate replica server for the recipient.

For instance, if an email is being sent from a@localhost to b@localhost, the SMTP server connects to the backend master node and requests the replica server for user b. The SMTP server then issues a PUT request to store the email on the appropriate replica server. This architecture ensures that emails received from external clients (like Thunderbird or telnet) are directly stored in the user's inbox, maintaining consistency and availability.

2.5 Web Storage Service

PennCloud's web storage service allows Users to upload files to their personal storage, which involves the frontend server parsing the content and sending it to the backend. The PUT command is used in the format

```
PUT user,/content/<folderPath>, base64EncodedValueOfFile
```

ensuring that file data is correctly encoded and stored.

The system's design supports efficient file retrieval and management. When a user wants to download a file, the frontend server sends a GET user,/content/¡folderPath¿ command to the backend, which responds with the file content. This process ensures that file access is quick and reliable.

The web storage service features a user-friendly interface that displays a list of files and folders Users can create, delete, and rename folders, move files and folders, and remove files as needed. The platform supports various file types, including text, PDFs, images, audio, and video, and can handle files of at least 50 MB in size. The system is designed to efficiently handle both text and binary files.

2.6 Admin Console

The Admin console provides control over the system's servers and storage nodes. It is a separate server from the frontend server and can be accessed via http://localhost:7000/admin. This interface allows administrators to enable or disable any frontend servers or backend storage nodes, providing fine-grained control over the system's operation.

The Admin console obtains the list of backend storage nodes by connecting to the backend Master Node through TCP Port 2 and requesting the "config.txt" file. This configuration file includes details about all storage nodes, which are then displayed in the admin console.

When enabling or disabling backend storage nodes, the admin server sends ENABLE/DISABLE commands directly to the specified storage node using TCP Port 2. This ensures that nodes can be managed without affecting other parts of the system. For frontend servers, the admin console sends a SIGINT to shut down a server or executes a system command (system("./frontendserver -v -p " + serverPort)) to start it.

The Admin console also allows administrators to view the contents of storage nodes by sending LIST commands to the specified node through TCP Port 2. This feature ensures that administrators can monitor and manage the data stored within the system, maintaining overall integrity and performance.

3 Major Design Decisions & Challenges

3.1 Key-Value Store

- **High-Level Overview:** Understanding how the backend is structured was essential. We designed it with multiple replica groups, each with primary and secondary servers to ensure redundancy and fault tolerance.

- **Crash Recovery:** Crash recovery was challenging as we did not initially plan for it. We had to devise a method using epoch values to gauge update timestamps, ensuring that data remains consistent and recoverable after a crash.

- **Additional LIST Command:** Supporting an extra command, LIST, which performs regular expression matching against the keys in the bigtable-like map, added complexity to our key-value store implementation.

- **Multi-Threaded File Locking:** Handling concurrent writes to the same file on the same server within a replica group required multi-threaded file locking. We solved this at the primary server level by blocking requests to the secondary server sequentially, but it was difficult to solve concurrency at the primary level itself.

- **Communication Challenges:** Communication within the backend and to/from the frontend was extremely challenging, as we had to manage four open ports simultaneously—two TCP and two UDP. Ensuring seamless and reliable communication required careful planning and implementation.

- **Server Role Identification:** Identifying primary servers and informing storage nodes about their roles (primary, secondary, or dead) involved significant string parsing and communication. This process was complex but crucial for maintaining system consistency and reliability.

3.2 Frontend Server & Load Balancer

- **Webpage Implementation:** Implementing the web pages and functionalities, including navigation and information submission, was non-trivial. We needed to properly design the HTML code and insert links to realize features like navigating and tracking folder locations.
- **File Uploading:** Parsing the content as multipart/form-data for file uploading was challenging. We used the boundary provided in the HTTP header to identify file information and actual data, ensuring correct processing of uploaded files.
- **Load Balancer Communication:** The frontend server needed to send heartbeat and client information to the load balancer via an additional UDP socket. The load balancer had to use locks for reading and editing server information to accurately redirect clients, ensuring balanced and efficient load distribution.

3.3 User Accounts

- **User Authentication:** Ensuring secure and efficient user authentication was challenging. We were originally not putting the passwords into base64 and couldn't handle special characters.
- **Session Management:** Maintaining user sessions to track active users and provide a seamless experience required careful planning. Ensuring that sessions were secure and did not lead to unauthorized access was a significant challenge.

3.4 Webmail Server

- **SMTP Integration:** Integrating the Simple Mail Transfer Protocol (SMTP) for sending emails required a robust design to ensure that the frontend server could clearly communicate with the SMTP server and the SMTP server could convey that info to the backend server.
- **User Interface Design:** Creating an intuitive and user-friendly interface for the webmail system was non-trivial. We needed to ensure that users could easily navigate their inbox, read, send, and manage emails.

3.5 Web Storage Service

- **File Upload and Download:** Handling file uploads and downloads efficiently was challenging. We needed to ensure that files were correctly parsed before going into the backend.
- **Folder Management:** Implementing features for creating, deleting, renaming, and moving folders required some thought for how to save folders and understand what info the frontend needed from the user to move the folder.

3.6 Admin Console

- **Server Management:** Enabling and disabling frontend and backend servers required a robust and secure mechanism. We had to ensure that these operations did not disrupt the system's overall functionality. The solution we came to was having a seperate TCP port just for the admin console.

4 Changes since Demo

- **Email Deleting:** Emails can be deleted
- **Server Status:** Server Status is Displayed in Admin Console
- **Admin Console Visuals:** Admin Console Visuals Improved
- **Folders:** Folders can be moved