CIS 5050 Project Report

King Long Tang, Jingqi Yao, Ruiwen Yang, Richard Zhang

1. Overview

This project implements a distributed cloud platform similar to Google Apps, providing webmail and storage services. The system architecture follows a multi-tiered design where stateless frontend servers interact with users through a web interface while all persistent data is maintained in a distributed key-value store backend. This separation ensures high availability and fault tolerance, allowing seamless user experience even when individual components fail. Our implementation focuses on three core design principles: scalability through data partitioning, reliability through replication, and availability through fault detection and recovery mechanisms.

2. Design Description

Our project contains the following components:

(1) Frontend web servers handle all the HTTP requests, authentication, and user interactions. The frontend servers would interact with the backend server to retrieve and update user data and use cookies to maintain session state. Multiple frontend instances can run simultaneously to distribute load. User account sign-ups, web storage service, and mail service can be handled and processed through the frontend.

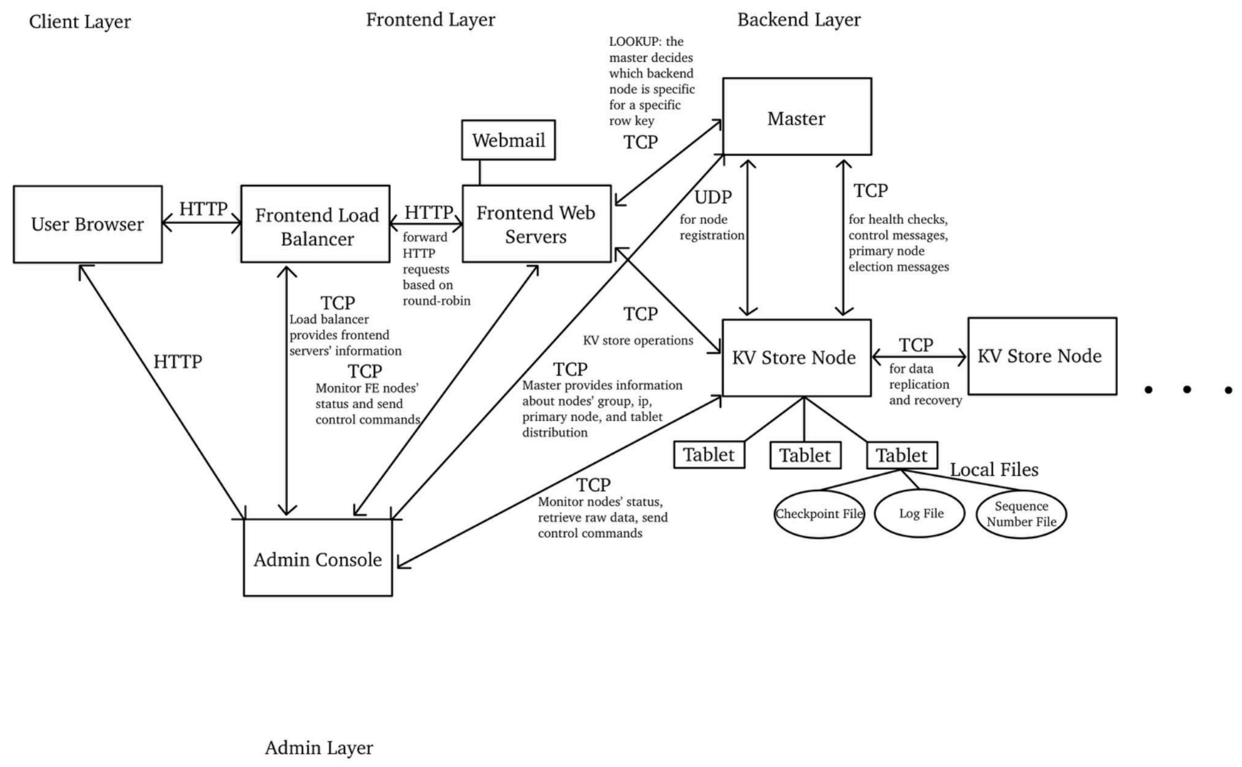
(2) Frontend Balancer distributes requests across frontend servers, ensuring even distribution and fault tolerance. To be more specific, it distributes incoming user requests across available frontend servers, providing a single entry point to the system.

(3) Backend nodes store all user data, including accounts information, emails, and files, using a distributed key-value store. Specifically, it is a distributed, partitioned, and replicated storage system that provides strong consistency and fault tolerance guarantees. The KV store is organized as a giant sparse table with rows and columns, supporting PUT, GET, DELETE, and CPUT operations.

(4) Backend master, which is the coordinator of backend nodes, elects primary nodes, manages tablet distribution, handles new node registration, handles LOOKUP requests from the frontend by locating the tablet that stores the corresponding row and returning the primary node information responsible for that tablet, and monitors their status. It contacts the admin console to give it information about all the backend nodes.

(5) The admin console provides a web-based interface for monitoring and managing the distributed key-value store system. Users can also view the raw data and tablet distribution here.

When a client first makes a request, the request will be directed to the load balancer, which helps to assign the request to an available frontend server according to current load statistics. The frontend server will then process the request according to its purposes. If the request is for login or storage, the frontend will query the backend to locate relevant data. If the request is for webmail services, then the frontend will forward the request to SMTP server for processing. Frontend servers will send periodic status updates to the load balancer so that it can make informed decisions on requests distribution. Admin console displays the status of backend nodes and frontend servers through constantly checking with them to detect failures.


Client Layer
Frontend Layer
Backend Layer
LOOKUP: the master decides which backend node is specific for a specific row key
TCP
Master
Webmail
User Browser
HTTP
Frontend Load Balancer
HTTP
forward HTTP requests based on round-robin
Frontend Web Servers
UDP
for node registration
TCP
for health checks, control messages, primary node election messages
HTTP
TCP
Load balancer provides frontend servers' information
TCP
Monitor FE nodes' status and send control commands
TCP
KV store operations
TCP
Master provides information about nodes' group, ip, primary node, and tablet distribution
KV Store Node
TCP
for data replication and recovery
KV Store Node
• • •
Tablet
Tablet
Tablet
Local Files
TCP
Monitor nodes' status, retrieve raw data, send control commands
Checkpoint File
Log File
Sequence Number File
Admin Console
Admin Layer

3. Features

Frontend:

(1) User Account & Sign up
User Accounts are supported where users can sign up, log in, manage their credentials, and access their webmail and storage services.

(2) Webmail Service
Webmail Service provides email functionalities, including sending, receiving, replying, deleting, and forwarding, from and to both mailboxes in the PennCloud as well as in other domains.

(3) Web Storage
Storage service enables users to manage files and hierarchical folder structures. Functionality includes uploading and downloading text and binary files, creating, deleting, renaming both files and folders, and moving these items between different folders.

(4) Load Balancer
Load Balancer distributes incoming client requests across available frontend server instances using a round-robin strategy. It monitors frontend server availability to ensure requests are routed only to responsive nodes.

Backend:

(1) Distributed Key-Value Store

The key space is divided into tablets based on row-key hashing, with each tablet assigned to three nodes. This method distributes data evenly across nodes.

(2) Replication & Consistency Mechanism
Each tablet has one primary node and multiple secondary nodes. Write operations must go through the primary node, which then propagates changes to secondaries. We also use WAL and periodic checkpoints to ensure data durability and consistency.

(3) Fault Detection & Data Recovery (See below for details)

(4) System Coordination & Primary Node Election
The master node maintains a registry of all backend nodes, assigns tablets based on a configuration, performs periodic health checks to detect failures, elects a new primary node, and sends a promote message to tell that node if the original primary node fails.

Admin Console:

(1) Backend Nodes & Frontend Servers Visualization
(2) Backend Nodes & Frontend Servers Shutdown/Restart Function
(3) Displays Stored Raw Data with Pagination

4. Major Design Decisions and Challenges

(1) Primary-Based Replication
To ensure fault-tolerance and data integrity, we will store every data tablet in three different backend nodes. If the primary node fails, the coordinator will set one of the secondary nodes to be the new primary node. We chose primary-based replication over quorum approaches for its simplicity and strong consistency guarantees. While quorum systems offer potentially better availability, the complexity of implementation and potential for inconsistencies led us to favor the primary-based approach. This decision simplified our system while still providing fault tolerance.

(2) Consistency Model
We implemented sequential consistency through the primary/backup approach to make sure that all updates to a given row are applied in the same order across all replicas, maintaining a consistent view of the data at all times.

(3) Checkpoint and Log-Based Data Recovery
A significant challenge was implementing efficient recovery without transferring excessive data. In our system, all the backend nodes would periodically go through log and checkpoints updates. Each storage server will have a checkpoint file and a log file so that servers can get back in sync with the others after a crash.
Specifically, we adopted the following strategies in data recovery. The system periodically does checkpoints (60 seconds) for every tablet in every backend node to capture system data. We use incremental logs to record operations between checkpoints. Each checkpoint has a version

number to determine optimal recovery strategy when a node is restarted. Each tablet of a node also has a sequence number file to record the sequence number of the latest operation in this tablet for data recovery purposes.

We use three strategies to minimize recovery time by transferring only necessary data during node restoration based on different scenarios. The recovery protocols include local recovery, where the first node to restart after all nodes in a replication group fail, so the restarted node recovers its data using its local checkpoints and logs; full synchronization, which transfers the primary node's checkpoint and full log when the recovering node's checkpoint version is outdated; and incremental log synchronization, which applies only new log entries from the primary when checkpoint versions match.

(4) Handling Concurrent Node Failures

One of the most challenging aspects was managing multiple concurrent node failures. We implemented a recovery priority system to ensure three things. First, when all nodes fail, the first to recover becomes the primary. Second, when a subset fails, the system maintains availability through remaining nodes. Lastly, recovery operations are sequenced to prevent version conflicts

(5) Consistency vs. Availability Tradeoffs

We prioritized consistency over availability in certain failure scenarios. Firstly, we ensured operations can only be performed in the primary node of a node group. Secondly, the data users upload is only considered written when replicated to all nodes. Lastly, recovery processes would temporarily pause write operations to ensure consistency.

(6) Concurrency Handling

All read and write operations are synchronized using mutexes to ensure data consistency and prevent race conditions when multiple clients access the same data at the same time.

(7) Stateless Frontend Architecture

Frontend servers were designed to be stateless, with all user session information, file metadata and file contents persisted directly in the backend KV store. This allows seamless redirection of clients to other frontend instances on server failure, thereby enhancing fault tolerance.

(8) Storage Service KV Store Schema

A two-table KV store schema was adopted for managing user file storage per user:

The <username>:map table tracks every absolute file or folder path as a column name. Each column value is a UID that points to the file's content in the data table.

The <username>:data table holds all file payloads. Column names correspond to UIDs, and each column value contains the file's content encoded in Base64. This ensures that binary data can be stored and transmitted safely as strings.

When a file is uploaded, the service first writes its path and newly generated UID into the map table. It then writes the same UID along with the Base64‑encoded content into the data table. To download a file, the service first reads the UID from the map table using the file's path, then retrieves and decodes the content from the data table. Renaming or moving a file requires only map‑table updates: the service looks up the existing UID by the old path, removes that entry, and creates a new entry under the new path with the same UID. Because the data table remains untouched, these metadata operations stay fast even for very large files.

(9) Load Balancer

A round-robin load balancer was implemented to distribute client requests across frontend server instances. This component actively monitors frontend server health to ensure traffic is routed only to responsive nodes.

(10) Web mail

Building on the codebase from Homework 2, we primarily changed the way emails are stored by integrating with a KV store backend. Several client classes were developed with extensive testing to facilitate communication between the POP3 and SMTP backends, the KV store backend, and the frontend. To enable sending emails to other domains, we implemented an MX record lookup tool that recursively resolves the host from a domain name.

(11) Other Implementation Challenges

Detecting node failures reliably in a distributed environment

Managing version conflicts during recovery operations

Ensuring checkpoint and log integrity during system crashes

Coordinating concurrent write operations across distributed nodes

5. Division of Labor

King Long: Responsible for frontend servers, login, storage service, the load balancer. He also fixed bugs in the integration of backend and frontend.

Ruiwen: Responsible for backend key-value store and the mechanism of recovery & fault tolerance

Richard: Responsible for backend key-value store, the mechanism of recovery & fault tolerance, and the shutdown/restart functionality of backend servers and raw data viewing in the admin console.

Jingqi: Responsible for user account management, the webmail function, and their integration with the frontend and backend.

All team members did testing, integration, and bug fixes together.