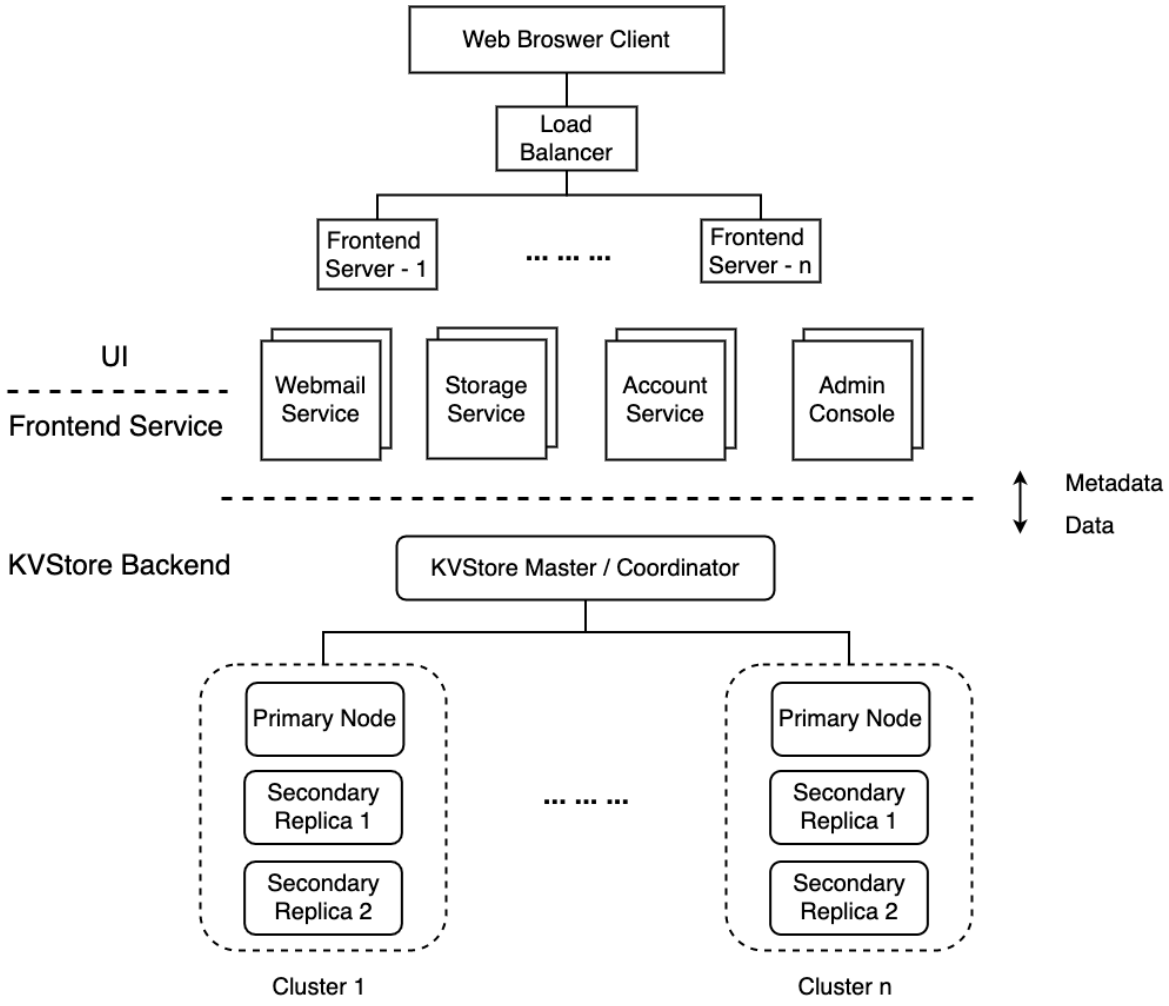
CIS 5050 PennCloud Report: Team 07

1. Big Picture

1.1 Architecture



1.2 Introduction

PennCloud provides users with Mail service and Storage Drive service. Mail service allows clients to send, forward, and reply to emails with local and outer users. Drive service enables users to upload and download all kinds of files(pdf, txt, mp4, etc.) and manage their files by creating, removing, renaming files and directories. All data including user information, mail, and drive files together with metadata are stored in a BigTable-like key-value storage(KVStore). Administrators can monitor the status of each service and check data in KVStore through the admin console.

Technically, Penncloud is a C++ project in support of multiple distributed system features such as load balance, fault tolerance, recovery, and consistency features. The front-end UI is written in React and backend communication is built upon the gRPC framework.

2. Overview of Features

2.1 Load balance

Loads on our system are balanced with a load balancing server, which redirects users to different frontend servers.

2.2 Authorization

Users can sign up, log in, change passwords, and log out. While in the application, authorization is checked every time via a cookie with a unique session ID. Users are forbidden from performing tasks and accessing data if the session id is invalid.

2.3 Mail

Users can check inboxes, check mail content, compose new emails, forward email, reply, and delete. Users can receive emails from thunderbird clients and send emails to users in our system.

2.4 Storage Drive

Users can check current drive contents, create folders, upload files, rename folders, rename files, move folders, move files, delete folders, delete files. Our system supports nested folders, and nested contents can be moved along with the folder itself. Our system also supports files of any kind, with file sizes up to 1GB (configurable to larger size).

2.5 Admin Console

Our admin console allows users to view the status of our servers, as well as suspending (stop incoming connections) the servers directly through our web interface. The console also allows users to view the raw data stored in the servers.

2.6 KVStore

Our backend storage is a distributed key-value store, somewhat analogous to Google's Bigtable. It supports PUT, GET, CPUT and DELETE operations. Different formats of files or content can be stored and accessed efficiently. There is a master node (coordinator) managing multiple clusters, each of which deals with requests of some specific range (implemented by hashing). Each cluster consists of several worker nodes, one of which is a primary node while others are replicas to provide fault tolerance. The worker nodes are entities to implement data storage and accessing, which manage multiple tablets, log files and checkpoints. We provide configurable parameters to set the number of tablets in memory and in disk, enabling users to tune the overhead of context switching and making our system flexible between performance and robustness. Delicate rules are applied for each operation under complex scenarios to guarantee consistency.

To make our implementation clear and scalable, we do not make different implementations for primary and secondary nodes. Only the master node designates the primary and their roles are distinguished by different requests - that is, their implementation shares the same code, but will accept different requests. The client gets the address of the primary node from master, either the client or master will directly send primary requests for primary nodes. When a primary node receives a request, except for doing the corresponding operations, it will

also send corresponding secondary requests to secondary nodes (health check before sending). The secondary nodes will follow the operations to ensure consistency.

2.7 Consistency and Fault tolerance

Our system provides consistency and fault tolerance, so that data will still be accessible when nodes crash. All these happen efficiently and silently - that is, users may even not notice the existence of crash or recovery.

All the data, including frontend services' metadata, are stored in KVStore, so that the crash of any frontend servers will not cause any data loss. The backend key-value store servers are also fault tolerant. Within the same cluster, raw data stored are replicated across different servers, with one designated as primary nodes. Shutting down one or more of the primary nodes and non primary nodes will not affect the operation of the services, as they can easily recover with the help of other alive nodes. The recovery process is efficient (we add checksum to avoid transferring duplicated data) and robust. The administrator could manually suspend and revive specific nodes in the admin console page. When node terminates services or crashes accidentally, it will also be recovered automatically once the master detects the node is accessible through a regular health check.

2.8 Extra credit

For extra credit, we implemented a beautifully designed user interface with an outstanding user experience. We also implemented the drive that allows storage for large files.

Support for large files in drive

Our implementation supports storing files larger than 10MB (up to 1GB) in drive.

Responsive interface

We use client-side JavaScript to render the web pages, which significantly reduced the need to fetch web pages from backend servers and improved the response time.

Streamlined authorization process

When the user is not logged in, they would be redirected to the login page if they try to access protected resources. All non-existence URLs would be directed to the home page.

Visual clue with colors and fonts

Our design helps the users to better navigate the system with colors and fonts that are informative in addition to the texts that carry that information. We used different colors for the buttons that carry out different functionalities in accordance with our general perception. Different font sizes and colors are used to signify different purposes.

Hints and prompts

We used different types of hints and prompts to guide users. For instance, when the user tries to delete a folder in drive that contains subdirectories and files, the interface would prompt the user to remove those contents first. When composing a new email, the interface automatically parses user input and adds different emails just as in modern Gmail and Outlook.

3. Design Decisions

3.1 Consistency and Fault tolerance

3.1.1 During normal execution

When KVStore nodes are alive and execute normally, we should make them ‘prepared’ for any potential crashes. The primary node is the major endpoint to deal with data requests from clients. For each cluster, we also have two (configurable) secondary replica nodes. Whenever the primary node receives a request to modify the tablet (PUT, CPUT, DELETE), it will notify secondary nodes to do the same operation. When it receives a GET request, even if such a result doesn’t modify tablets, it will check if it will result in tablet switching between memory and disk (by default configuration, we have 5 tablets in disk and 3 of them are loaded into memory for accessing). If so, it will also notify secondary nodes to do tablet switching so that all the three nodes share the same memory and disk state, which will make further recovery reliable and fast.

Each tablet also has a corresponding log file and checkpoint file in disk. Any modification operations including PUT and DELETE will be appended to the log file. When a tablet is switched from memory to checkpoint due to the loading of other tablets, its checkpoint will be updated to the latest state in memory and the log file will be cleared. At any point of time, if a tablet is in memory for accessing, the following equation always holds: memory state = checkpoint state + operations in log file.

3.1.2 Suspend or Crash

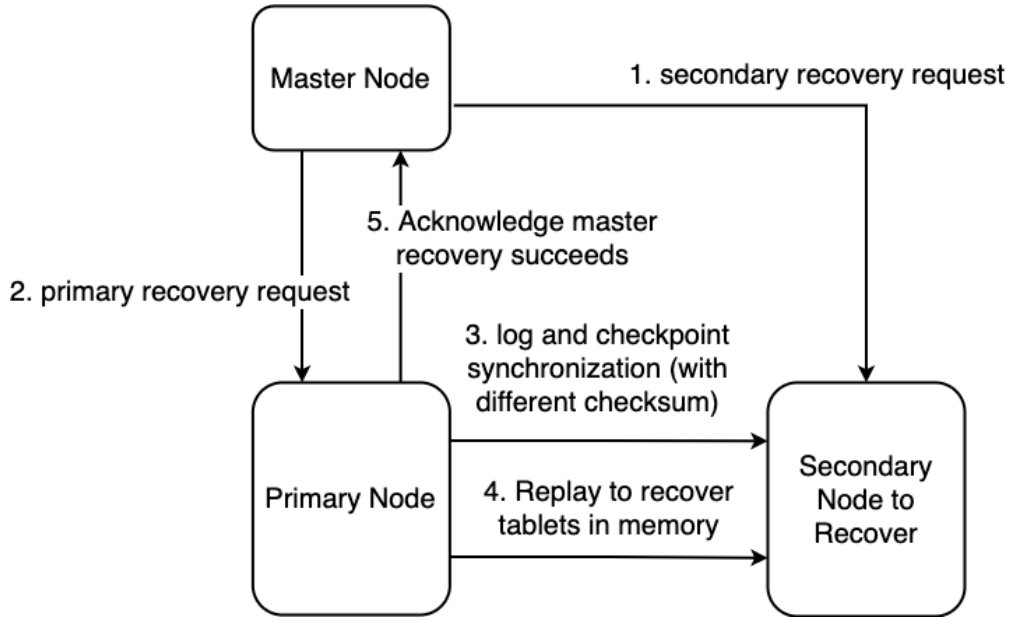
There are two kinds of ‘crash’ and we provide fault tolerance for both of the two cases, either it happens on a non-primary node or primary node (under such a case, master will select a new primary to guarantee there is always a alive primary node).

The first case is SUSPEND, which is manually triggered by the admin console. When the KVStore master receives a SUSPEND request from the admin console, it will forward it to the designated node to suspend. When the node receives it, it will change its status from RUNNING to SUSPENDED but do not terminate itself in the backend. During SUSPENDED status, it will reject any incoming requests except for REVIVE. The primary node will firstly do a health check for its secondary peers before sending requests, and send requests only to the peers who successfully respond with a RUNNING status.

The second case is CRASH. It happens when a user intentionally terminates the node in the backend with a Ctrl-C, or the node crashes accidentally. Under this case, the regular health check by master and pre-request health check by primary will fail and this node will be marked as CRASHED by the master.

3.1.3 Recovery

If a node is previously suspended by the admin console, it can be recovered when the admin console triggers a REVIVE request. If it previously crashes and entirely terminates, once it restarts, it will be noticed by the master node by regular health checking (per second), and the master will automatically send a recovery request to it. Under both cases, the master will send a secondary recovery request to the node to be recovered (It’s guaranteed the current primary node is alive, because when a primary node is suspended/crashed, master will select another primary node).



When the secondary node receives a secondary recovery request, it will update its status to RECOVERING, so that it will only receive recovery-related communication with the primary.

The master node then sends a primary recovery request to the primary node, asking the primary node to assist the secondary with recovering. The primary node starts a new thread to do recovery so that the master won't be blocked by waiting for the recovery to finish. It also sets its status to RECOVERING and will reject any other unrelated requests.

It then follows with the communication between the primary node and the designated secondary node to recover. The first step is to synchronize log files and checkpoints for each tablet. For each file, the primary will first send its checksum, and the secondary node responds with its own checksum. Only when the two checksums are different, it means the secondary's file is deprecated, and it should be synchronized with the primary's with file transfer. We use checksum to avoid unnecessary file transferring. At the end of this step, all the checkpoints and log files are synchronized.

After all the files are synchronized, the primary node sends a replay request with the sequence number of its current tablets in memory to the secondary node. The secondary node's memory state may either be empty (from crash) or deprecated (from suspend). We mentioned to ensure consistency, such an equation must hold for tablets in memory: memory state = checkpoint state + operations in log file. For each tablet in the primary node's memory, the secondary node will first load it from checkpoint to memory, and then replay the operation according to its log file. When the replay step finishes, the secondary node's memory state is also synchronized, which means it has been fully recovered.

At last, both the secondary and primary nodes update their status to RUNNING. The primary node acknowledges the master node that the recovery for this secondary node has been finished. Then they can execute as normal.

3.2 Nodes management

The KVStore consists of one coordinator(master node interchangeable) and multiple clusters where each cluster consists of multiple replications. The master node plays a critical role in nodes' communication.

3.2.1 Status Maintenance

The master node maintains the status of all replication nodes in clusters. The status can be one of RUNNING, SUSPENDED, RECOVERING, and CRASHED. RUNNING means the node is healthy and running properly. SUSPENDED means that the admin console suspends this node and this node will not accept any more requests from clients except for “Revive”. RECOVERING means the node is communicating with a primary node to get the latest data. CRASHED means that the node program exits and can only be restarted from the terminal.

3.2.2 Health Check

Every second the coordinator sends a “CheckHealth” message to all nodes and changes the status of each node accordingly. The coordinator updates the status booklet based on “CheckHealth” responses. If no response is received(RPC failed), then we consider the node to be crashed. If the “CheckHealth” message indicates that the node is running again after CRASHED, then the coordinate also needs to send a special message to the primary node to start recovering.

3.2.3 Primary Node Election

The master node records the primary node of each cluster. Whenever clients need to send a request to KVStore, the coordinator will hash on “row” and “col” values and find the corresponding cluster, and then relay the request to the primary node of that cluster. When a node is crashed or suspended by the admin console, the leader will take a round-robin strategy to find the next running node and designates that node as the primary node.

3.2.4 Console Information Retrieval

The coordinator also communicates with the admin console and provides three APIs, “PollStatus”, “Suspend” and “Revive”. The “PollStatus” method returns the status of all nodes managed by the coordinator. The “Suspend” method suspends a specific RUNNINg node and the “Revive” method revives a specific SUSPENDED node and also notifies the primary node to do recovery.

3.3 Large file

It needs several modifications when we want large files to go through our frontend.

3.3.1 Double loop HTTP request loading

Our socket reading method in homework is keeping reading and then scanning for line feeders. This won’t work when sending large files because we can’t load all binary strings if we set the buffer too small, and can swallow the next request if setting the buffer too big.

To solve this problem, we used the double loop loading and used a separate buffer in each loop.

1) In the first loop we keep loading a small trunk of data into buffer 1 from the socket.
2) In the second loop we read line by line into buffer 2 from buffer 1, break until finding an empty line. If buffer 1 is exhausted, call loop 1.

We know we have the complete header after reading the empty line. Then we search for `Content-Length` keyword to know the length of the request body, thus neither load a portion of large files, nor swallow the next request.

3.3.2 Metadata in url

Normally it's easy to parse the metadata from the request body for small files. However, parsing metadata in large files can make the frontend server slow and under heavy memory pressure. It would even crash sometimes if we set a small memory limit for our virtual machine. To solve it, we send the metadata of large files directly in the request url to avoid body parsing.

3.4 Storage strategy

KV Store	Password(string)	Cookie(url encoded unix time)	Files(json object)	Mails(json object)	FileId 1(json object)	FileId 2(json object)	…
User A							
User B							
…							

We have four columns for each user: password, cookie, files, and mails.

1) Mails are small text files so we save the whole mail list into one cell for fast loading and storing.
2) Json is naturally tree-structured data-interchange format, which is ideal for file path storage. When we create, move, and delete files and folders, we just need to read the `Files` cell, modify it and save it back.
3) We just save the `fileId` in the `Files` json object. When actually uploading their content, we create a separate cell for them. They can be large files so we try to avoid loading and storing them unless necessary.

3.5 User interface

Our system uses the React.js framework as our frontend user interface. React is responsive and renders more quickly than other frameworks and vanilla JavaScript and HTML. We also included Bootstrap as our styling library. For communication between backend and frontend, we used a set of carefully designed Restful API and packed information in JSON format for easy parsing on server's end.

4. Distribution of Work

Zhouyang Fang: 1) Server side API handling. 2) HTTP server. 3) Webmail service. 4) Load balancer.

Jiaxuan Ren: 1) Rest API Design, 2) user interface, 3) client side API calls and business logics, 4) Storage service.

Yao Shen: 1) KVStore worker node-side implementations 2) KVStore PUT/CPUT/GET/DELETE implementations, admin console node-side implementations 3) Replicas, consistency, inter-node communications and data storage 4) Design and implementations for fault tolerance and recovery.

Shidong Lyu: 1) gRPC framework, 2) KVStore master node implementation, 3) KVStore client which exposes PUT/CPUT/GET/DELETE methods for front-end service to use, 4) Admin Console backend part includes polling status, handling "suspend"/"revive" from the admin console.