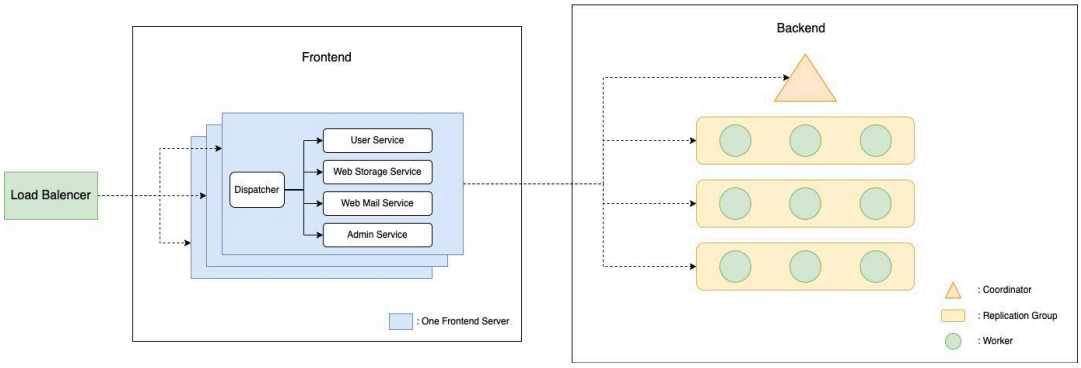
PennCloud Project Report

Design Description

On the high-level, our project is composed of two parts: frontend and backend.
Each frontend server handles the request from the user, follows different service logics, and generates corresponding html pages.
Backend is composed of one coordinator and several worker nodes, providing the key-value pair storage service.
Frontend server communicates with the backend server through the TCP connection.
We also implemented a load balancer that monitors the health status of the frontend nodes and randomly redirects the user request to one of the frontend nodes. There is also an admin service that monitors the health status of frontend nodes and backend worker nodes, and can stop and start any backend worker node.
Below is a diagram of our overall system architecture.



Implemented Features

In this section, we will list the features we implemented in each component, following the order specified in the project handout. At the end of this section, we also include the extra credit features we implemented.

Key-value Store (Backend)

Persistency

Data, the key-value pairs, is persistently stored to the disk.

Duplication

Rows are distributed among all replication groups.
Each row is replicated within a replication group, which contains one primary and two secondary nodes.

Consistency

Consistency is maintained throughout the replication by sequentializing the requests at the primary using sequence number, centralized checkpointing, and row-level locking.
Each worker node supports both local recovery, by reading the checkpoint file and log files, and optimized remote recovery, by trying to transfer the log file difference before transferring the whole checkpoint file.

Fault Tolerance

The system provides fault tolerance by supporting all operations and maintaining consistency at least one node (any one node) in one replication group is active. In our case, each replication group has a size of 3.
The system handles the node crashing at any time, including during any operations or during replication forwarding. This is achieved by enabling worker nodes to report other inactive worker nodes to the coordinator during their communications.

Cache

Cached parts of the local key-value pairs are in memory for fast reads.

Frontend Server

Framework

Implemented Web server based on MVC design, supporting user-defined handlers, models, and views. Constructed HTTP request parser and HTTP response generator, supporting various requests including chunked transfer encoding, various responses, including JSON and chunked transfer encoding, and persistent connections. Implemented request dispatcher based on path requested and method used.

Authentication and Session Management

Implemented filter and authentication based on JWT in cookies for user isolation and management. All succeeding connections with the JWT token as part of the cookie would be considered as requests from the same user until the JWT token expires.

Frontend Pages

Built user interfaces for storage service and mail service based on Tailwind CSS, including client-side JavaScript logic. Implemented a HTML template engine and View resolver, generating HTML content based on Models created in handlers.

User Service

Designed and implemented the business logic for user-related sections and user-specific access control, including sign-in, sign-up, reset password, and issuing and validating JWT.

Webmail Service

Implemented an SMTP server to accept emails from both our system and outside our system; an SMTP client as service to send emails to both local and remote users (SEAS email accounts). Designed a web mail user interface for all email operations.

Web Storage Service

Implemented all necessary storage service business logic, mimicking the Linux commands, including uploading/downloading files, creating directory, moving the directory/file, renaming the folder/file, and deleting the directory/file.
Construed the chunked strategy to chunk the large file into a bunch of chunked data, improving the fault tolerance and balancing the data in backend KVS.
Utilized the Base64 encoding and decoding to support binary files.

Designed and Implemented the Inode to store the folder structure/metadata for files, decoupling the actual file data and file structure for better performance and scalability.
Implemented CRUD functionality in DAO layer and Model Object serialization/deserialization to support mapping from C++ class object to actual data storage in Key-Value Storage.

Admin Console

Designed and implemented the business logic for Monitoring the whole system, including accessing the current frontend/backend node status(alive or dead), displaying the current key-value pairs inside the alive nodes, and up/down the backend nodes.
Implemented the timed refresh in HTML script to have a better user experience.

Load Balancer

Implemented a load balancer that takes initial http requests and redirects them to one of the frontend nodes. The load balancer is also responsible for collecting health status of frontend nodes for admin service.

Extra Credit

The maximum size of the file that our Web Storage Service supports is 150 MB.
Webmail service supports separate Inbox and Sent (like folders).

Design Decisions

In this section, we will discuss some design decisions we made in our design in every component, following the order specified in the project handout.

Key-value Store (Backend)

Replication

Our system implemented the primary-based replication with remote writes, where each replication group has three nodes (one primary and two secondary nodes). Each server node is either a primary or a secondary node (but not both) for a certain set of rows based on their row key hash values. The client can reach out to any worker node in a replication group, and if a secondary node receives a write operation, it forwards the operation request to the primary and then forwards the response from the primary to the client.

Checkpoint

We implemented a centralized checkpointing. The primary node in a replication group keeps the number of write operations and will do the checkpoint if the number of writes reaches a limit. The primary then sends out the checkpoint signal to other secondary nodes and other secondary nodes will follow the signal to do the checkpoint.

Sequentialization

When a primary forwards a request to a secondary, either the request is for the replication or for the checkpoint, the primary will attach a sequence number to the message sending to the secondary. Each secondary node maintains a message queue and only delivers the message in the queue if the sequence number of the message is continuous.

This will prevent the out-of-order message arrival caused by separate TCP connections from the primary to the secondary nodes.
In the discussion with the professor, we realized that this sequence number approach may limit the throughput of the write operations as each write operation needs to access and write to the sequence number stored in the primary nodes with read-write consistency.

Cache in Memory

Each worker node, either the primary or the secondary, has a cache containing a subset of the cells it stores in the disk. Each write operation will first write to the log file and then update the cache in the memory. This will help to increase the speed of reading data as there is no disk reading involved for the cells stored in the cache. We have a limit on the number of cells stored in the memory cache, and if the number of cells reaches that limit, a randomly selected cell will be picked from the cache to remove.

Frontend Server

MVC-based Web Server Framework

To support the need for the display of dynamically loaded data from the user, isolation between business logic and interaction with the user based on MVC architecture is created for encapsulation and scalability.
Several components are created for different purposes:
Connection Listener: Accept the connection from the client's browser, establish the TCP connection, parse the request header of the request from the client each time.
Request Parser: Parse the request body, supporting multipart form, chunked transfer encoding, and JSON.
Filter: Filter out invalid requests, including requests without valid JWT and requests accessing forbidden paths.
Request Handlers: Define business logic for different scenarios and functions, processing the request, generating models for the response.
Request Dispatcher: Assign appropriate handlers based on the request’s path and method
ViewResolver: Resolve the view templates based on the response’s resources.
ResponseGenerator: Generate html web pages based on view templates and models from the handler.
Messenger: Generate final response, choose the appropriate content to return from the response generated by the generator, supporting returning files, HTML web pages, JSON objects, and plain texts. Write the final response stream to the connection back to the client.

RESTful Frontend Design

All user interactions are written in HTML and JavaScript in the frontend web pages. The decoupling between the user interface and the backend logic is nicely done. Storage service, webmail service, and user services adopt the RESTful architecture. All modifications to the drive are done by sending JSON requests and receiving JSON responses between the web pages and frontend web server, reducing the workload of rendering from the frontend web server.

Webmail Service

Webmail service, as an SMTP client, is responsible for sending messages to both local and remote SMTP servers. To connect the SEAS email server outside of the school's network, a VPN is necessary.
While each frontend node has its webmail service, we have a standalone SMTP server running between all frontend nodes and backend nodes. This decision is made since receiving messages is only a small part of the webmail service, and we don’t bother each frontend node to have its own SMTP server.
The mailbox of each user is split into Inbox and Sent. When a user sends a message, it first goes into the Sent box, then the SMTP server receives the message and stores it in Inbox.

In KVS, the messages as well as the metadata are stored per row per user. To keep track of the messages, two metadata files (for Inbox and Sent) are created to store the message_id as well as to cache the subject and date which are shown in the email list, therefore we can just check the metadata files to retrieve the data for the email list.

Web Storage Service

MVC design

To decouple the business logic from the data-access logic, we added an extra DAO layer to fetch data from the backend, realizing the Model layer. We designed the data schema and serialization/deserialization in the Model layer to map the database and C++ class object.

File Storage

Given that each key-value cell cannot exceed 1MB, we need to have a policy to chunk the large file into a bunch of small chunked data to support files that cannot fit memory. Each time the frontend receives the large file, it will be cached into the frontend server disk. The service layer will read the file and save the file into the backend KVS chunk by chunk, with the assigned chunk ID. Assigning one file to one node is reasonable, assuming that one file can fit the disk in one backend node. In this case, we significantly reduce the overhead of fetching the IP and Port of the target backend worker node with further reduction by caching the fetching result.
Another requirement is to support the Binary File. By default, when we read the binary file and save it into character, we will have some special characters like CRLF, causing a disaster when using `string`, which will be used all around our system. The solution will use Encoding/Decoding to encode the binary data into a `string`.

Folder Structure

To modularize and decouple the file system structure and actual file, we designed a tree-like folder structure containing `metadata` called `Inode`. Each time the metadata or folder structure changes, we must modify the whole `Inode` structure and save it to the backend. But we don't need to change the Inode when the actual file changes. Our design could be better by decoupling the `metadata` with the folder structure with the assigned UUID as the key of the folder/file node. In that case, we don't need to modify the whole `Inode` tree when the folder name or `metadata` changes, which has a greater scale and parallelism.

Admin Console

Update Strategy

Since our design naturally inherits the two layers, coordinator and workers, it is natural to design a two-layer monitor system, which scales better than one layer and updates timely.
In our two-layer architecture, the first layer is the coordinator part(Load balancer for the frontend nodes and backend coordinator for backend nodes), which uses the Push-based method to get information from all nodes. The second layer is the centralized monitor system, which utilizes the Pull-based method to fetch data from the frontend/backend coordinator.

Load Balancer

When a frontend node starts, it sends a registration request to the load balancer so that further user requests can be redirected to it. When a user request comes in, the load balancer checks in cycle for the next available (can establish

a connection) frontend node and responds with a 307 Temporary Redirect, or 500 Server Internal Error if no frontend node is available.
To respond to the admin service's request, the load balancer tries to establish a connection between each registered frontend node and return the corresponding status.
The load balancer is not the web traffic bottleneck since it does the redirection, so the further data will not pass through the load balancer.

Challenges

In this section, we will talk about some challenges when implementing some components, following the order specified in the project handout.

Key-value Store (Backend)

Fault Tolerance Any Time

One of the challenges in the Key-value store is handling the fault tolerance.
The first challenge is that the node can crash anytime during the communication, including every request forwarding, remote writes, checkpoint forwarding. This means that in addition to the heartbeat detection, we need another way to detect the inactive nodes. In our implementation, we check the communication status before and after message sending and receiving to check if the receiving node is active or not. If the receiving node is not active, the sending node can report the inactive node to the coordinator and the coordinator will update the replication group status and assign a new primary if needed. Through this way, our system is able to handle the fault tolerance anytime during the operation, making our system more robust when there is a node crash.

Debugging

Another challenge in Key-value store implementation is debugging. There are many edge cases when considering fault tolerance and it is very hard to reproduce the bug we encountered after stopping one of the worker nodes. We need to read through and analyze the logs (not the data log but printing log) of different workers and figure out what happened and find out the potential bugs and then add more logging if needed.

Frontend Server

Debugging

Debugging needs to overcome the unpredictable behaviors between different web browsers, especially when uploading and downloading large files.

Webmail Service

Since we need to support out-of-system messages, the service needs to follow standard SMTP protocol. Different from what we have in homeworks, this includes proper header fields with special formats, which takes some time referring to the protocol and Thunderbird.

Web Storage Service

Design logic for file system

The challenge of designing the web storage service is the design of the file system. How to decouple the folder structure with actual data and metadata. The naive version will be all things coupled together. However, we decided to learn from the Linux file system and use `Inode` to decouple the folder structure and metadata from the actual file, reducing the overhead when the file changes.
Even though we decouple the actual file, further decoupling the metadata from the folder structure would be better. Therefore, it is really a challenge to design a well-formed architecture for file systems. If we have more time to do it, we will definitely find a better solution and implement it.

Labor division

- Haoze Wu: Frontend Web Server, Frontend Webserver Handlers, Webpages
- Xuanbiao Zhu: Storage Service, User Service, Admin Service
- Weihao Xia: Webmail Service, Load Balancer
- Beilei He: Key-value Store (Backend)

Code Version

Changes after Demo

After the demo, we added more comments in the backend and added the instruction to build and run both the frontend and backend services. We also added five screenshots of our application.

Version used for Demo

The hash of the last commit before the demo is `4c12c05bec61e585c9c88a664195dea7e7ecdf79` in the branch `dev`.