CIS 5050 Final Project Report: Team T21

Ally Kim, Ben Jiang, Kathryn Chen, Michael Yao; and Xuanbiao Zhu (Lead TA)
{allykim, bjiang1, fdshfg, myao2199}@seas.upenn.edu

Design Overview

Our system broadly consists of (1) **PennCloud Email:** an email system where users can send emails to both external and PennCloud users, and receive emails from PennCloud users; and (2) **PennCloud Drive:** a file storage system where users can upload files and store them for later use. Both components make use of stateless frontend servers, a frontend load balancer, a backend coordinator node, and backend storage servers.

Briefly, communication between the frontend and backend servers is coordinated through the master backend coordinator node. When a frontend server wants to execute a command, it first contacts the coordinator to ask for the address of a storage server in the relevant partition. After obtaining the address from the coordinator, the frontend server submits the command to the relevant backend server. The backend servers perform the operation and return a response to the frontend. The aforementioned communication steps are implemented via stream sockets using the Transfer Control Protocol (TCP).

The backend coordinator is connected to every backend storage server in order to keep track of active servers and current primaries. The identities of the backend servers are determined at compile time, while the availabilities are determined at runtime through heartbeats. Separately, the frontend servers send periodic heartbeats to the master frontend load balancer. The identities and availability of the frontend servers are determined dynamically at runtime.

A separate SMTP server handles all requests for delivering emails. The server supports local clients such as Thunderbird and Telnet, client requests to deliver emails to PennCloud inboxes (i.e., `@penncloud.com`), and client requests to deliver emails to external inboxes (i.e., `@seas.penn.edu`).

Finally, an admin console displays the current status, IP address, and port of both frontend and backend servers. Data stored in the backend servers can also be accessed read-only via the admin console. The admin console also enables a privileged user to shut down and restart backend servers.

1. PennCloud Mail

Responsible Team Member(s): Ben Jiang (`bjiang1`)

1.1. Overview

PennCloud Email allows users to read, compose, reply, forward, and delete emails. Users can create an account with an email address using the `@penncloud.com` domain and login to view their inbox. Each email address is unique, meaning that no two users may have the email username. The uniqueness of email addresses is enforced when new users sign up for the PennCloud service. Emails in the inbox are shown with their subject and timestamp (time sent). Users are able to view individual emails and associated metadata, such as the email sender, subject, timestamp, and body of the email. Users are also able to reply, forward, or delete emails.

1.2. Components

PennCloud Email is implemented using two core components: (1) Javascript custom code included in relevant HTML files (`frontend/templates/email_compose.html` and `frontend/templates/email_inbox.html`) to allow clients to communicate with frontend servers; and (2) appropriate request handlers implemented by the frontend servers.

1.2.1. Javascript Component

All client-side communication is implemented in custom Javascript code loaded together with relevant HTML files. Client navigation between different PennCloud pages is made available in the left sidebar.

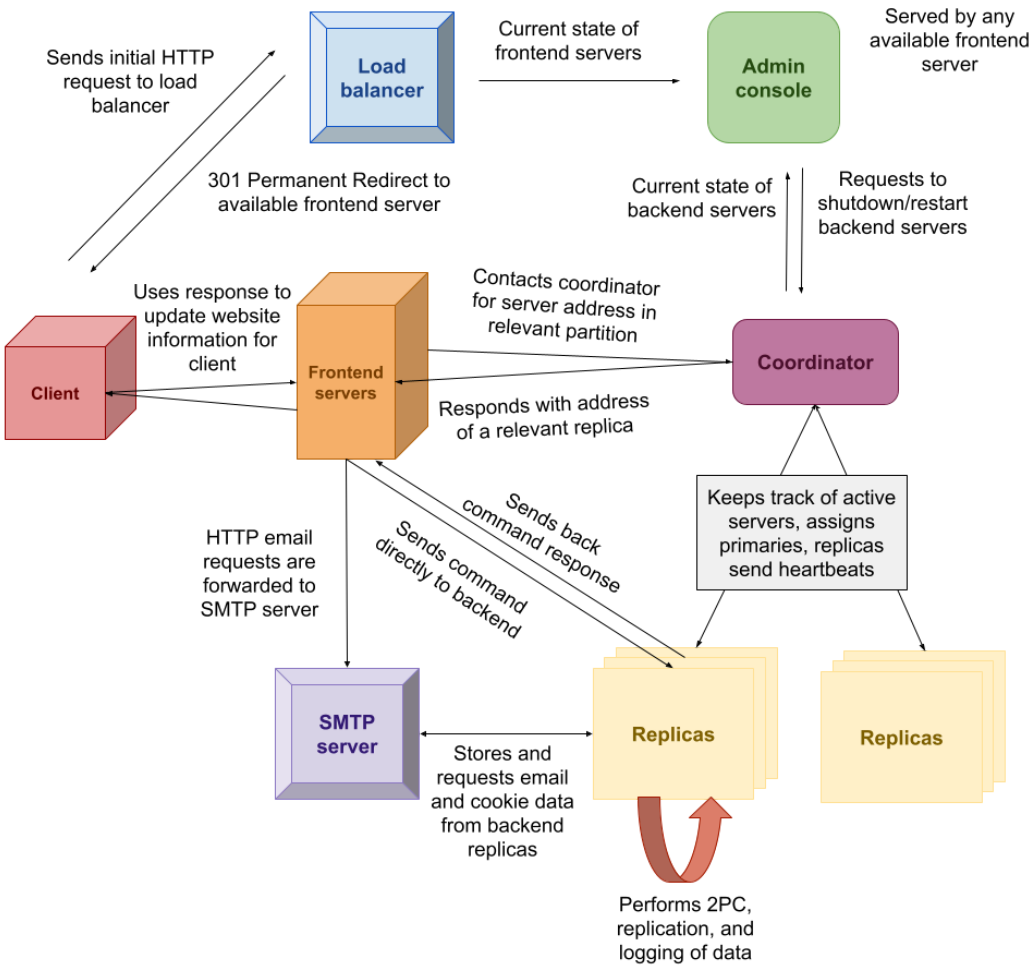

Figure 1: Diagram of Overall Architecture and Implementation.

`Compose email` redirects the user to `/email_compose`, where the client is able to specify an email recipient, subject, and message body to send an email. When users have filled out all the fields and click send, the email is sent via a POST request to the frontend HTTP server. Our implementation supports sending emails to both internal users (i.e., `@penncloud.com`) and to any email within the `@seas.upenn.edu` domain. When an email is sent to an internal user, the frontend server instructs the backend server to generate an email object, hash it, turn the data into a binary string, and store the string in the backend. For each user, a vector `eid_vector` of email hashes keeps track of the emails contained in the user's inbox. The recipient's `eid_vector` is updated accordingly when a PennCloud user sends an email to the recipient.

`Inbox` retrieves the user's email and displays a list of the user's email to the client. A PennCloud user's inbox can be manually retrieved to show more recent emails by reloading the webpage. Each email in the inbox is an HTML `<li>` element with the subject and timestamp; clicking on a particular email redirects the client to a separate page to (1) view the particular email of interest; (2) reply or forward the email; or (3) delete the email.

`Home` returns the client to their homepage to access the PennCloud Email or PennCloud Drive systems.

1.2.2. Frontend HTTP Server Component

Inside the frontend server, there are several routes set up to handle requests sent by Javascript. We implement HTTP endpoints for (1) GETting the emails in the user's inbox (`/email-list`); (2) PUTting a new email in a user's inbox

(`/send-email`), and DELETEing an email (`/delete-email`). When sending an email, the request will be forwarded to an SMTP server, which allows us to send emails to external domains (i.e., `@seas.upenn.edu`) and for local clients (e.g., Thunderbird) to send emails to PennCloud users.

2. PennCloud Drive

Responsible Team Member(s): Ben Jiang (`bjiang1`)

2.1. Overview

PennCloud Drive enables users to upload and download files to a replicated file storage system. Users are also able to create folders; our system supports a nested folder structure. PennCloud users can rename, move, and delete both files and folders. All folder and file names must be unique within a given directory. Our implementation of PennCloud Drive supports fast and persistent file uploads and downloads for up to 10 MB. All file types are supported.

2.2. Components

PennCloud Drive is implemented using two core components: (1) Javascript custom code included in relevant HTML files (`frontend/templates/files.html`) to allow clients to communicate with frontend servers; and (2) appropriate request handlers implemented by the frontend servers.

2.2.1. Javascript Component

Beside the upload button is a new folder button, which the user can use to create a new folder. When creating a folder or file, our implementation requires that the user does not have duplicate file or folder names within any given directory; this requirement is enforced client-side. The new folder will show up as the list of files is dynamically reloaded even after the HTML page itself is loaded. The file display section displays all files and folders in the user's drive.

For every file, the user can download, rename, move, or delete it. When downloading the file, the frontend server GETs the file data from the storage; files that are at least 10 MB in total size are automatically downloaded via chunked-transfer encoding per RFC 2616. The user will then be able to view their downloaded file in their download folder.

Similarly, all folders can be opened, renamed, moved, or deleted. Folder opening is implemented via a GET HTTP request, and the client's web browser will be redirected to display the folder's content. Deleting the folder recursively deletes every file and every folder inside (i.e., equivalent to `rm -rf` on a Linux command line) before deleting the folder itself. Files can be renamed and moved to other folders, taking the absolute path of the destination folder as input. Our implementation supports files up to 10MB in size; in our hands, uploading a 10 MB file takes 10 seconds, and downloading a 10 MB file takes 25 seconds.

2.2.2. Frontend HTTP Server Component

We implement two key HTTP GET request handlers in our HTTP frontend server: (1) `/files-list` to list the contents of a folder; (2) `/download` to download a particular file. The specific folder to view and file to download are taken as GET URL parameters. Folders are stored as keys by backend servers according to their absolute path, and map to vector values of the corresponding all files (absolute paths) and folders inside the folder. Files are stored in the backend by absolute path and file data (converted into a binary string). Auxiliary GET request handlers are implemented for moving, renaming, deleting, uploading, and creating new folders.

3. Frontend Components

Responsible Team Member(s): Michael Yao (`myao2199`)

3.1. Load Balancer and Server Membership

Our implementation makes use of a single, centralized load balancer that listens to incoming requests from HTTP clients on port 8080. The load balancer also simultaneously listens for datagram heartbeats from frontend HTTP servers; a node identifies itself as a frontend HTTP server by sending heartbeats every six seconds to the load balancer. If a heartbeat has not been received by a particular server for at least two consecutive heartbeat intervals, then the load balancer assumes that the particular HTTP server is no longer responsive and will no longer redirect incoming traffic to that server. In this fashion, frontend servers are able to dynamically join and leave the network of frontend servers, and membership of frontend servers is determined dynamically at run time. The current state of all frontend servers is recorded and made available to admin console users, which is detailed in the **Admin Console** section below.

If a frontend server becomes unavailable after a client has already made previous requests to that frontend server within the same session, the client can navigate back to the load balancer (i.e., port 8080) to be redirected to an alternative frontend server.

3.2. Implemented HTTP Methods

The following HTTP 1.1 methods are implemented according to RFC 2616: `GET`, `POST`, `DELETE`, `HEAD`, and `OPTIONS`. Privileged paths that require clients to be authenticated (i.e., must have first logged in via the `/` root path) will redirect the user to the homepage (i.e., `/` root path) if an authentication is invalid or not provided. User authentication is implemented via HTTP cookies according to RFC 6265; please see the subsection on **Session Management and User Authentication** below for additional details.

3.3. Session Management and User Authentication

PennCloud users are asked to login via their PennCloud email address and password at the root domain. User passwords are hashed via the standard 256-bit Secure Hash Algorithm (SHA-256) using the OpenSSL C++ library implementation before being stored in by backend servers. Upon successful login, session-persistent cookies containing (1) a unique 128-byte randomly generated alphanumeric session ID; (2) the client's username email address; and (3) the client's first and last name are returned to the client. The client uses these cookies to authenticate subsequent privileged HTTP requests (i.e., reading the user's emails or uploading private files): the session ID must match the session ID for the client that was stored by the backend servers at the original time of password authentication. When a user logs out of PennCloud or changes their password, then the client is instructed to erase all aforementioned cookies from the session. Notably, our method of user authentication does not require statefulness of any particular frontend HTTP server, and allows multiple users to be signed in to PennCloud at the same time.

3.4. Chunked Transfer Encoding

PennCloud frontend HTTP servers are able to both read and write chunked transfer encoded message bodies according to RFC 2616. By default, PennCloud file download requests for files that are at least 10 MB in size will be downloaded using chunked transfer encoding in chunks of 64 KB.

4. Backend Components

4.1. Table Storage

Responsible Team Member(s): Kathryn Chen (`fdshfg`)

The backend table supports the GET, PUT, CPUT, and DELETE commands. The storage table in memory is represented as an unordered map with the hash of an `std::pair` of the row and key values, mapping to the value stored at that specific row and column. To prevent race conditions, each existing row and column pair also has its own set of `pthread_mutex_lock`'s, the existence of which is created and destroyed along with the table entry.

When a GET command requests the value from a table entry, the entry's read lock is acquired. For the other three commands (i.e., PUT, CPUT, DELETE), the server acquires the write lock. On each backend server, each partition is

further divided into five tablets. The data is divided among the tablets based on the sum of the ASCII values of each character in the row string modulo 5.

Tablets are loaded into and cleared from memory based on their need and the amount of current available memory in the system. Upon server initialization, tablets are loaded into memory until they take up to half of the initial available memory. If a read or write operation needs to access a tablet that is not loaded, the required tablet will be loaded into memory, and other tablets will be dynamically deallocated as necessary. When a tablet is cleared, a checkpoint for it is taken. Logging and checkpointing are also implemented on an individual tablet basis.

The specifics of processes regarding fault tolerance, checkpointing, and consistency are detailed below.

4.2. Coordinator

Responsible Team Member(s): Ally Kim (`allykim`)

Our backend architecture leverages a coordinator server to keep track of relevant information regarding storage nodes and their mappings to specific tablets, which is required to allow for easy communication with frontend servers. It is also used to assign and keep track of primary servers for a given partition, and to communicate server information to the admin console.

One of the main roles of the coordinator is to keep track of active nodes. When a backend node becomes active, it notifies the coordinator, which begins monitoring its heartbeat messages. The coordinator continues to receive heartbeat messages from each storage node every second, and uses these messages to check which nodes are still alive. If a node stops sending heartbeat messages, the coordinator records that the node has stopped working.

The coordinator also handles server partitioning and assigning primaries by reading an input text file of server addresses and their partitions, and then assigns the role of primary to a server in a partition if there is not already a primary. If the primary for a partition goes down or is shutdown by the admin console, the coordinator assigns another active server as primary if one exists for the partition. If not, the coordinator waits until a server in that partition becomes active again. Whenever a new primary is chosen, the coordinator broadcasts the primary address to all replicas within the partition to enable primary-replica communication. The coordinator also creates partitions based on the number of partitions specified in the input text file. It dynamically divides the key space based on the number of partitions: the first letter of the username modulo the number of partitions determines the partition number the key is assigned to. When a frontend server attempts to read from or write to the file, it first contacts the coordinator with a "`HELO user email`" message , and the coordinator responds with an address from the partition that is designated for the key-value pair.

4.3. 2PC and Primary Replication

Responsible Team Member(s): Ally Kim (`allykim`; 2PC) and Kathryn Chen (`fdshfg`; Primary Replication)

The backend servers use a two-phase commit (2PC) consensus protocol and primary-based replication for the replication protocol. When a non-primary replica receives a valid command from a frontend server, it immediately forwards it to the primary. The replica also creates a unique hash for the command based on the command, row, column, and contents (if any) as well as a `msg_queue` object, which it stores in an unordered map mapping to the command's hash. Once the primary receives a valid command, it begins the 2PC and the replication process. The primary server also creates a hash for the command and stores it locally in its own map.

The initial stage of 2PC begins with the primary assigning a transaction number to the command, which is used to serialize all incoming commands and ensure sequential consistency. It then logs the command in its 2PC log file and forwards the original command with the hash prepended to all replicas in the partition in the form of a PREPARE message. When a replica receives a PREPARE message from the primary, it also logs the PREPARE message, parses the message for the hash, and–if it has not already–creates its own `msg_queue` object mapped to the hash. All servers in the partition, including the primary, place the transaction in an ordered queue, where ongoing transactions are saved and, once ready, executed sequentially by the transaction number.

The replica then attempts to acquire the lock for the row and column pair for the forwarded command. If it succeeds, it responds with a "`YES`." Otherwise, it responds with a "`NO`". Once the primary receives all responses from replicas, the primary moves to the commit phase if all responses are "`YES`" and the primary is also able to acquire the relevant locks. Otherwise, the primary aborts the transaction. The primary logs the transaction decision in its 2PC log file. The replicas receive the `COMMIT` or `ABORT` message from the primary, which they log and respond to accordingly. Once the server that has originally received the command from the frontend server executes the transaction, it responds back to the frontend server with the result of the transaction.

Of note, because the 2PC and primary replication message components are split by colon literals, we impose a requirement that email addresses for users or filenames for files cannot contain colon literals in our implementation.

4.4. Logging, Checkpointing, and Recovery

Responsible Team Member(s): Kathryn Chen (`fdshfg`; Logging, Checkpointing, Recovery), Ally Kim (`allykim`; Recovery)

The storage servers use a distributed checkpoint system to ensure that the checkpoint is successful and the cuts are consistent across all replicas. At the start of a server process, the server recovers its table data by parsing the checkpoint and then log files of all loaded tablets. It then receives the primary message from the coordinator, and if it is not assigned the role of primary, it contacts the actual primary and requests the log files for all of its tablets. This is because the current primary is always the node in the partition that has been running for the longest, and will therefore have the most comprehensive and recent log files. The replica server then parses the logfiles the primary sends: if the primary's log file starts at a more recent checkpoint, it clears its own log file and copies everything from the primary log file. If the checkpoint numbers are the same, the replica only starts copying from the log entry number that is greater than their largest log entry number. Additionally, whenever a new server has been chosen as the new primary, the server parses its TPC log to learn about any transactions that may have been interrupted before completion. Once the new primary is connected to all the current replicas in the partition, it aborts all pending transactions by sending an ABORT message to all the replicas.

When a transaction is committed, it is logged by all replicas of the relevant partition, meaning the transaction is pushed onto the log queue for the relevant tablet. Every three seconds, the queue is flushed, meaning all transactions in the queue are written to the tablet's log file. Each transaction is given a monotonically increasing log number that is reset when a new checkpoint begins. When a log file reaches a certain size, the current primary starts a distributed checkpoint. It sends a message to all replicas, and all storage servers for the partition begin checkpointing. This involves clearing the tablet's checkpoint file, writing the contents of the table currently in memory to said file, increasing the checkpoint number, and clearing the tablet log file. While checkpointing, in order to ensure the cuts are consistent across all replicas, outgoing messages are queued and new logs are not written to the logfile until the entire process is finished. Under the right circumstances, multiple checkpoints can occur simultaneously for different tablets.

5. Admin Console

Responsible Team Member(s): Ally Kim (`allykim`) and Michael Yao (`myao2199`)

We implement a special webpage (`GET /admin HTTP/1.1`) accessible through any frontend HTTP server to visualize privileged information about our system and both shutdown and restart backend storage nodes. The admin page requires the privileged user to authenticate for access, which is implemented via providing a `password` URL parameter equal to `cis5050`. The admin console shows all frontend and backend nodes in the system and their current states (i.e., alive or unresponsive), and also allows users to view raw data in the storage containers.

To retrieve the status of individual frontend servers, the HTTP frontend server serving the admin console client makes a request to the load balancer to return a list of all of the frontend servers that have previously sent at least one heartbeat datagram to the load balancer since the start of the load balancer process. Frontend servers that have sent a heartbeat at least once in the last 2 heartbeat intervals are marked as available, and all other frontend servers are marked as unavailable.

Admin console users can also disable or restart individual storage nodes. We implement a "virtual" disabling of storage nodes: namely, if an admin console user requests to disable a particular backend storage node, the frontend HTTP server relays this request to the backend coordinator node, which then sends a special blocking marker message to the specified storage node to close all frontend connections. The coordinator node marks the node as paused and ceases to forward any frontend server to the node until it receives a request from the admin console to restart the node. Although the paused storage node is prevented from handling any frontend connections, it remains connected to the primary node to allow for data replication. Once the admin console resumes a given node, the coordinator unmarks the node as paused and starts forwarding frontend nodes to the server again.

6. Major Design Decisions and Challenges

Responsible Team Member(s): Ally Kim (`allykim`), Ben Jiang (`bjiang1`), Kathryn Chen (`fdshfg`), and Michael Yao (`myao2199`)

One major design decision was to use single, centralized nodes (e.g., frontend load balancer, backend coordinator, primary-based replication, etc) in our PennCloud system, as opposed to implementing fully distributed alternative solutions. This approach enabled us to implement and debug our system more effectively and efficiently, but at the potential cost of introducing singular points of potential failure and bottlenecks. Future work may explore implementing fully distributed algorithms if we seek to scale our PennCloud system to a larger user base.

Separately, while we did not have to handle the cases where the coordinator goes down, a major challenge was figuring out how to handle cases in which a primary unexpectedly shut down in the middle of an operation. Because it takes time for a new primary to be elected, the election of a new primary in the middle of an ongoing transaction results in some difficult edge cases. For example, we needed to figure out how a replica that was elected to be the new primary in the middle of primary-based replication should handle the remainder of the transaction.

Another challenge was that since two-phase commit requires backend nodes to send multiple messages to each other, this inevitably increases the amount of time it takes to perform a write operation. When calling `PUT` or `CPUT` on large data queries, we needed to figure out how to efficiently handle several rounds of communication while maintaining replication and consistency. Ultimately, we optimized the process of reading and handling messages by increasing the number of bytes storage servers read at once and avoiding copying of large strings whenever possible. We also ensured that writing to and recovering from the logfile, unloading tablets, and checkpointing could be done concurrently.