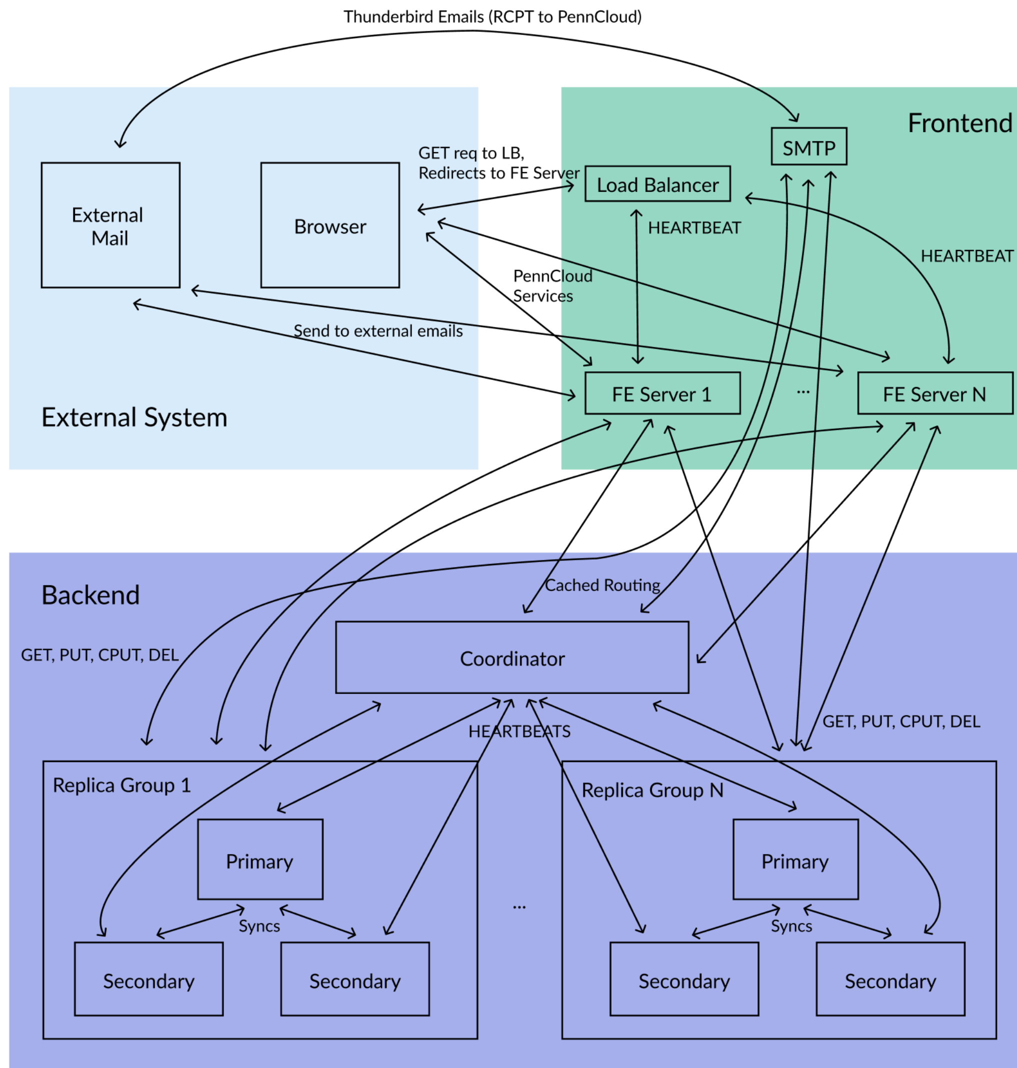
Team 03 CIS 5050 PennCloud Final Report

Angelina Hu, Grace Deng, Sam Wang, Seth Sukboontip

PennCloud is a platform made from a distributed frontend and a distributed backend. The backend is in charge of data management, which uses a simplified version of Google's BigTable. The frontend is able to interact with the row-key, column, and value storage via four main operations: PUT, GET, CPUT, and DELETE. Through these basic operations, the frontend renders the user interface and handles the business logic (computations, validation, formatting, etc) before storing the data in our backend storage.

PennCloud mimics the basic features of Google Drive and Gmail; it allows users to upload and download files as well as send and receive mail from within the PennCloud ecosystem and externally as well. Furthermore, we also implemented Blackjack as a minigame that relies on our backend storage and the distributed nature of our system. The final PennCloud system is also deployed on AWS via EC2 instances, allowing users outside of the local development machines to access our program.



Frontend and Services Overview

Frontend Webservers (Grace Deng) and Load Balancer (Sam Wang, Seth Sukboontip)

PennCloud uses multiple frontend servers to not only distribute the server loads but also create redundancy when a server fails. The frontend servers do not interact with each other. Rather, they interact with the frontend load balancer; they send periodic heartbeat messages to the load balancer to convey liveness and how many active connections each server is handling.

When a client wants to connect to the PennCloud system, they can connect via the load balancer's port, which would redirect the client to the frontend server with the least load. The balancer keeps track of the least loaded node by using a lazy heap with lazy deletion. The load balancer also interacts with the admin console to determine alive and killed servers and to send KILL and REVIVE commands.

Admin Console (Angelina Hu, Sam Wang, Seth Sukboontip)

The admin console lives at "\admin" and renders a dashboard for visibility and control over the cluster by using the "STAT" command to query the backend coordinator and the frontend load balancer. The console displays each backend node (group, ID, role, alive/dead) by using the "GET_MAP" command to query the coordinator for the virtual bucket routing table (tablets, responsible primary address, and group each bucket is assigned to). Moreover, it displays each frontend server (node ID, address, load, alive/dead) with the "STAT command". Nodes are listed deterministically. Additionally, the admin console supports killing and reviving frontend and backend nodes via special "KILL" and "REVIVE" control commands routed through the coordinator/load balancer for the backend/frontend, respectively, so that the admin console is never directly routed to the actual cluster.

User Accounts (Grace Deng)

PennCloud offers support for multiple users with concurrent access on persistent sessions. Users can create password-protected accounts, and session persistence is implemented using a browser cookie that is deleted upon logout. All of a user's data will have a row key that is prefixed with the username, which is used to hash row keys such that a single user's data can be accessed with some measure of locality with respect to the backend server nodes.

Mail Services (Seth Sukboontip, Sam Wang)

Our PennCloud system supports both internal and external communication via email. Within the system, users can send emails to each other using the mail user interface. Users can also receive emails from external emails through the SMTP server; in our case, for simplicity, we've opted to only receive emails from the localhost using Thunderbird. Moreover, users are able to send emails to other domains, such as a Penn email account by querying the recipient's domain MX records.

For each email, the user can forward, reply to, or delete the email. However, if the email was from a localhost domain (from Thunderbird), we disabled the reply option, as the system does not allow for sending to a localhost domain. For running the Thunderbird client, we reuse the POP3 server from HW2.

When an email is sent, including replies and forwards (they're just normal emails with specially formatted bodies), a copy is created for the sender, and another copy is created for the recipient. The duplication in the KV store allows the recipient to delete the email from their inbox, but the original sender will still see the email in their sent folder. The email data is stored on a single row with various columns like mail_id, sender, recipient, time, subject, body, etc. Moreover, each user has a mail metadata row that stores all of the existing emails in this specific user's mailbox, including both sent and received mail. This row simply has one corresponding column that lists all of the mail_ids in the mail box. Upon

loading the mail page, the system queries all of the mail_ids from the column to populate the inbox and sent folder. All creations or deletions must be reflected in the row, as it is the source of truth.

Webstorage Service (Grace Deng)

Each user has access to the web storage service in the form of a single-user, hierarchical filesystem. The system supports uploading files, creating directories, moving directories and files, renaming directories and files, deleting files, recursively deleting directories, and downloading files. Files of any type with sizes up to 50 MB are supported. Items in the file system each have a unique ID, which is used in the item's row key. Each item has a metadata row that stores its name, the ID of its parent, and whether it is a directory or file. Directories have an additional row that stores a list of child IDs. This allows us to move and rename items while only making O(1) row accesses.

When deleting directories, we can traverse through the child items and recursively delete the child content first. File content is chunked into packets, which are stored on unique row keys, which allows the content to split across tablets in the backend, in order to support files too large to fit into one tablet. Each packet contains at most 64 KB. We transmit the download to the user using chunked transfer encoding.

Backend Overview

KV Servers and Coordinator (All Team Members)

The backend is a distributed key-value store where all values are binary-safe and length-encoded across the wire transmission protocol, WAL Logs, and disk checkpoints. Each node maintains an in-memory tablet cache, which lazily loads tablets to memory on first access and uses an LRU policy for eviction. On eviction, a full disk checkpoint happens atomically using a copy into a .tmp and fdatasync.

We maintain row granularity locking for concurrent CRUD operat/rions using a five-lock hierarchy acquired in fixed order to prevent deadlocking and enable a release-early pattern for throughput: larger locks are acquired and held for minimal time before being demoted into finer-grained locks when possible. We use g_pq_mutex (node-level, always short-lived and held independently) and g_store_mutex (node-level, tablet map structure) -> eviction_mutex (tablet-level, tablet; held as a shared-lock for CRUD to prevent the tablet from being evicted mid-operation) -> registry_mutex (tablet-level, tablet row-col map) -> row_lock (row-level).

Within each group, the primary forwards every successful write to all secondaries using a fire-and-forget "REPLICATE" command loop to prevent I/O blocking and responds "+OK" to the clients asynchronously from secondary acks. Lagging secondaries will be caught up via checkpoint syncing upon rejoining/startup. Direct writes to secondaries are rejected via "REDIRECT" to the frontend request.

The coordinator is metadata-only and hands off all necessary information to the frontend server on first or stale refresh "LOOKUP". Failures can be detected by the coordinator via fast path (TCP disconnect marks node dead without timeout) or via slow path (watchdog polls every 200 ms and declares dead after 800 ms silence. The watchdog is also responsible for sending "PROMOTE_PRIMARY" / "NEW_PRIMARY", and upon restart, a node will sync by sending the coordinator "WHOAMI" before accepting connections again. The syncing process checks the checkpoint number on the current primary. If the local checkpoint number is less than the primaries, the new tablets. Regardless of the checkpoint number, the logs are always retrieved from the primary in the replication group.

Hashing, Virtual Buckets, and Splitting (Angelina Hu, Sam Wang)

We use three levels of indirection for our routing scheme by deterministically normalizing and hashing raw row keys into placement keys for virtual buckets and then dynamically assigning virtual buckets to groups and tablets (tracked in the coordinator map). We use num_virtual_buckets >>

fanout_buckets (number of tablets available to each node) to ensure proper balancing of user object data, but maintain that user metadata (raw row key in format <user> or <user>:<namespace>, i.e., "alice:mail") is always mapped to one bucket to optimize for locality with commonly used user metadata. This scheme allows us to dynamically split tablets without rehashing keys and distribute user data loads across tablets in exchange for losing some locality and increasing state tracking work.

Specifically, during each split, 50% of the virtual buckets in the tablet being split are moved to a new tablet. During the period that the virtual buckets are being moved, those virtual buckets are in a list of in-flight bucket IDs. Any operations to these buckets are blocked by a condition variable that notifies after saving both the new dest and src tablet to disk. Future requests from the frontend can use a stale tablet ID since the connection cache is not updated. To reduce unneeded load on the coordinator (since splits keep the data on the same replication group), the kv_server stores a map mapping from virtual bucket to owner that contains these updated ownerships, making sure that any operations to a stale tablet are transparently directed to the right location after splits.

Logging and Checkpointing (All Team Members)

Every write is appended to the per-table WAL log before the in-memory map is updated, and then fdatasync is called. The WAL is truncated after successful checkpoints, so recovery always replays only the changes since the last flush. The WAL also includes split records, enabling recovery from crashes mid-split. It is automatically loaded and replayed on every tablet startup.

We implement a 2PC protocol through a background checkpoint thread on the primary that periodically, every 10 seconds, sends a "CHECKPOINT_PREPARE" command to all secondaries to begin triggering a checkpoint. Secondaries that are mid-split will send back "-ERR". Otherwise, the primary increments its monotonic checkpoint_num and sends a "CHECKPOINT COMMIT", which triggers the checkpointing on the secondaries. This simplifies secondary recovery since a rejoining node will fetch the primary's checkpoint and WAL files via bulk transfer protocol using "GET_CHECKPOINTS", "GET_LOGS" upon finding a higher checkpoint_num.

Extra Credit Features

Blackjack Game (Grace Deng)

PennCloud also offers a multiplayer turn-based game, Blackjack. Users can access the Blackjack lobby, where any user can create a "table" that other players can join. Multiple tables may be available simultaneously, but each user may only participate in one table at a time. Tables can support 1-6 players. Each table has a "host", who will have the option to start the game and advance each round. The creator of the table is assigned as the host on creation. After the host starts the game, new players can no longer be added to the table. Players are assigned a "seat number" (starting at 1 and incrementing as new players join), and all players start with 1000 credits.

Each round starts with the betting phase, where each player takes a turn making a bet, starting from the lowest seat number to the highest seat number. Once every player has made their bet, the decision phase begins. The "dealer" (automated) deals everyone two cards and themself two cards. Each player gets a turn to "stand" or "hit", starting from the lowest seat number to the highest seat number. Players that chose to "hit" will be continually given the opportunity to make a decision until all players have either stood or busted. Once all players have stood or busted, the dealer will perform their automated behavior (hit until hand value 17). Then the round results are revealed, and balance changes are applied.

Players who end a round with 0 credits will be removed from the game at the start of the next round. The host can then choose to advance the game to the next round or end the game. If the host is removed from the game, the next lowest seat number player will be selected as the next host. Seat

numbers are persistent throughout the entirety of the game. If all players have been removed from the game, the game will be deleted. Blackjack tables are stored under a row key with the “blackjack” prefix, not specific to any user. Each table tracks metadata, including the turn number, the round number, the phase (start, betting, decisions, or results), and the deck (a list of cards from which the first card is removed for each deal, and shuffled and repopulated when empty). The metadata also includes seat-specific metadata, like the player name, balance, hand, and game status (whether they are “done” with their decisions or have “lost”). Each user also has a user-specific state under a row key with the user’s prefix, which tracks whether the user is currently in a game and which game and seat they are at, in order to enforce that each user may only be at one table at a time and may not accidentally access games that they are not a part of.

AWS Deployment (Sam Wang)

We deployed our system onto t3.medium instances. All of the processes are on separate instances, except HTTP Server 2 and one of the KV servers: AWS learner lab only allows 9 concurrent EC2 instances, so one node is shared. To allow for TCP communication, we created a security group where all group members were allowed to communicate through any port using TCP. External nodes were only allowed to connect through TCP for ports 7000, 8080, and 8100, for the load balancer and HTTP servers.

Major Design Decisions and Challenges

Replication Design: Primary Secondary Replication with Fire and Forget

We chose to use primary-secondary replication over read and write quorums. Having a single node as the primary ensured there was a clear canonical version of the data for the replication group, while the secondaries made sure we could handle the KV node crashes. We used a fire-and-forget message for replication: after the primary successfully completes a write operation, it sends a replication message to the secondaries, then acknowledges the HTTP client. This means there is a window where the secondary is behind the primary and has an inconsistent state. Due to this, we disabled reads from secondaries. Additionally, if the primary dies in the middle of sending the replicate messages, there will be an inconsistency between different secondaries that will not be resolved until the inconsistent servers restart.

Hashing and Virtual Bucket Design

Our row key design is <username>:<service>:<item_id>, with separate rows for each chunk (per the TAs’ suggestion). These are then translated into virtual buckets by hashing the postfix of the username, then taking the modulo of the fanout buckets. This design has a few cons. First, by including the username directly, we are required to restrict to safe usernames. It also makes it very costly to rename a user, so this functionality is not provided. Secondly, the number of fanout buckets places a cap on the amount of data a user can have: specifically, roughly 1 GB for the current parameter of fanout buckets (8). Tuning this parameter is a tradeoff because increasing the fanout buckets increases capacity, but also means that users with less data will have less locality since they will fan out across more tablets. Additionally, we only implement tablet splitting, not merging, so tablets that split but don’t have that much data continue to span multiple tablets.

Some more minor design improvements we could make would be making the servers event-based instead of thread per connection to improve the scalability. We also could use more consistent synchronization methods across the codebase: we use atomics, unique_locks, mutexes, semaphores, and condition variables across the codebase.