CIS 5050 PennCloud Report: Team 01

Yichun Cao, Chen Fan, Yazhe Feng, Yinda Zhang
Mentor TA: Yifan Cai
{ycao4,cfan3,yazhefen,yindaz}@seas.upenn.edu

ABSTRACT

In this report, we show the design of our final project. In §1, we first illustrate an overview of the architecture. Then, we describe our implementation of each component respectively. Specifically, we show the implementation of master (§2), frontend (§3), and backend (§4). Finally, we list the contributions of each team member to different components (§5).

1 ARCHITECTURE

As shown in Figure 2, our PennCloud project has 3 major parts.

- **Master server:** The master server is responsible for monitoring and load balancing. Specifically, it should communicate with every other server to track their status and kill/restart them when needed. In addition, it redirects client requests to the frontend servers and assigns backend servers to frontend requests.
- **Frontend servers:** The frontend servers are responsible for managing HTTP sessions and handling client authorization, storage service, and webmail service. Specifically, each frontend server should parse the requests from clients, read/write information from/to backend servers, and render pages back to clients.
- **Backend servers:** The backend servers are responsible for storing information from frontend servers, which provide key-value storage similar to that of the Bigtable [1]. Note that the data in different backend servers should keep consistent, and the key-value store should be resilient to node failures.

2 MASTER DESIGN

This section shows the implementation of the master server. We describe how the master server monitors other servers in §2.1, how the master server load balances client requests and frontend requests in §2.1, and how the master server is connected to the admin console in §2.3.

2.1 Monitoring

To monitor other servers, the master uses a specific port (*e.g.*, 10000) to listen to the connection from other servers.

Frontend servers: The master server keeps track of the state of connection of frontend servers. Once the frontend server starts a connection to the master, the master will add it to the frontend server list. Conversely, if the frontend server breaks the connection to the master, the master will delete it from the frontend server list.

Backend servers (Figure 1): We set three states of backend servers (*Alive*, *Crash*, and *Dead*) for better monitoring. In our system, there is a configuration file that contains the IPs and port numbers of all backend servers. The master first reads the configuration file to determine the potential backend servers and sets their states to *Crash*. Once a backend server connects to the master, the master will mark it as *Alive*. If the admin kills an alive backend, the master will mark it as *Dead*. The main difference between *Dead* and *Crash* is

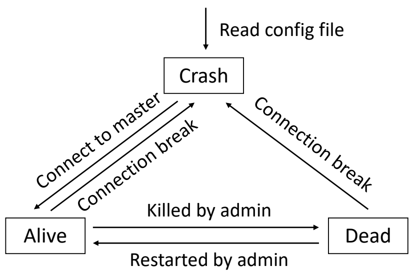

Figure 1: State transition of backend servers

that a *Dead* server should still keep the connection to the master so that it can be restarted. In addition, if the state of a backend server is changed, the master updates the information of the corresponding cluster (*e.g.*, re-select the primary) and broadcasts it to all alive backend servers in the same cluster.

2.2 Load balance

In our design, the master server creates a separate thread using a different port (*e.g.*, 10001) from that of monitoring to work on load balancing.

Client requests: The client only needs to know the address of the load balance. When the client tries to connect to the load balance thread, the thread will use the HTTP 302 command [2] to redirect the client to an active frontend server. Specifically, because the master server knows the status of every frontend server, it can randomly choose an active frontend server and return its address to the client.

Frontend requests: If the frontend server wants to read/write data of a user, it will ask the master for the corresponding backend server. The master server returns the address of the primary backend server according to the username. Note that all data of the same user will be recorded in the same group of backend servers. In our design, the master server hashes users to different groups of backend servers.[1]

2.3 Admin console

The master server would create a thread that awaits browser requests from the admin console page via a specific URL (*e.g.*, http://localhost:8600/admin). The administrator can view all alive frontend servers, the current status of all backend servers, and all raw data of a given user when requested. The admin console also supports operations such as killing an alive backend node and restarting a dead one. All requests are sent through HTTP POST requests, and the master server parses each request, sends corresponding commands to the requested backend server, and broadcasts CLUSTER messages to other backend servers, explained further in §4.2.

[1]Our design does not support dynamic membership for now.

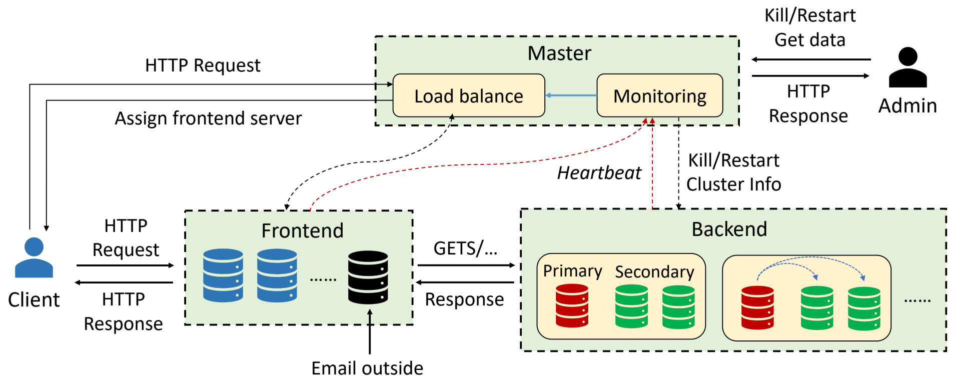

Figure 2: Architecture

3 FRONTEND DESIGN

This section shows the implementation of frontend servers. We describe how frontend servers interact with others (*e.g.*, browsers, master, and backend servers) (§3.1), manage HTTP sessions (§3.2), and handle client authentication (§3.3), storage service (§3.4), and webmail service (§3.5).

3.1 Interaction with others

Clients: The frontend server communicates with clients using HTTP requests and HTTP responses. Frontend servers identify operation types depending on the contents of HTTP requests. The HTTP requests are stored in the Request class, and frontend servers communicate with backend servers to generate a response with the corresponding HTML page containing data from backend servers.

As shown in Figure 3, to communicate with master and backend servers, we use *protobuf* [3] to serialize the request to string. All inter-server communications use TCP. We add a 64-bit (fixed) integer at the beginning of the string to show the size of the message. When receiving a message from other servers, the server reads the first 8 bytes to get the size, reads the following string, and uses *ParseFromString* to extract the fields we need.

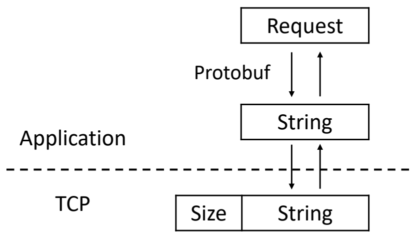

Figure 3: Inter-server communication

Master: Initially, the frontend server should send a message to the master to let it know that it is alive and can receive requests from clients. In addition, as shown in §2.2, the frontend server will communicate with the master server to get the address of the primary backend server of a user after receiving requests from clients.

Backend: The frontend server uses kv_command class to communicate with backend servers. After receiving the address of the primary backend server from the master, the frontend server will identify the operation types from the request, call the corresponding operation (shown in Table 2), and wait for the return value.

3.2 Manage HTTP sessions

The frontend server is able to handle GET/POST requests from clients.

GET request: For the GET request, the frontend server often renders HTML/CSS/JS pages in order to display dynamic data to the user. The main pages that we have include: login page, sign up page, home page, error page, admin console, email service, storage service, *etc.*

POST request: For the POST request, the client pushes necessary information (*e.g.*, file, password) into the HTTP message body, and the frontend server will parse the HTTP request to get the information.

3.3 User authentication

The frontend server allows users to sign up, log in, and change passwords. The password is recorded in the key-value store. Once the user logs in successfully, the frontend server will generate a random number as the cookie and record the mapping from the number to the username. Then every time the frontend server receives a request, it will get the username based on the cookie and mapping. If the request has an unknown cookie or no cookie at all, it can only GET the login page or sign-up page. In addition, the frontend server periodically clears the mapping to reduce memory usage.

3.4 Storage service

The frontend server provides users with a simple drive storage service, with support of upload/download file, create/view folder, and delete/rename/move for both. Each user is assigned a root folder (named using username) upon registration, containing all the files and folders created. Folder hierarchy is maintained through metadata only, with no additional information kept in the key-value store. Every file uploaded is assigned a unique, monotonically increasing ID, which is used as the storing key for easy management. Implementation details are discussed as follows.

Metadata: The metadata is mainly responsible for tracking the mapping from file name to the unique ID, and folder name and content. The ID assigned to each file will be used as the key for

	Name	Things to update
Folder	Create	Metadata
	Move	Metadata
	Rename	Metadata
	Delete	Metadata and Files
File	Upload	Metadata and Files
	Move	Metadata
	Rename	Metadata
	Delete	Metadata and Files
	Download	No update

Table 1: Operations provided in storage service.

storing file content in KV store, which simplifies the implementation for file rename/remove. All folder information is tracked in the metadata. When a folder is created, we add an entry in the metadata, which includes the folder's complete path, ID assigned, and each directory entry's mapping from its name (excludes parent path) to ID. Updates in metadata are written back to KV storage immediately using CPUT to ensure consistency.

Upload file: Each file is assigned a unique ID when uploaded. The mapping from the file name (not the complete path) to its ID is stored as an entry of its parent directory in the metadata. The ID assigned is used as the key for storing the file content in KV storage. Requests to upload files with invalid names will be rejected.

Download file: The storage service will query the file content from backend servers and transmit the information to users through HTTP responses.

Create folder: When a folder is created, we will add a new entry for tracking the name, ID, and content for the new folder, and also update the parent directory information in the metadata.

Move file: File move is implemented by updating the directory entries recorded in the metadata for the previous and new folders. Files can move to any folder that the user created. If a file is moved to its original folder, this operation has no effect.

Move folder: Implementation for moving folders is similar to files, except that folder cannot be moved into its child folders. During folder move, only valid destination folders are offered to the user.

Rename: As we assign each file/folder a unique ID as the key, renaming simply requires updating the name recorded in the name-to-ID mapping, no need to make updates to the Key-value storage.

Delete: Deleted file/folder information will be removed from the metadata, and requests to remove file content will be sent to backend servers.

Update metadata: As mentioned earlier, metadata is modified and written back to KV store immediately using CPUT after each update to ensure consistency. If metadata has changed while processing the current request, CPUT would fail. Careful consideration would be required for file upload and deletion, as metadata CPUT could be unsuccessful after KV store is already updated. In our implementation, the file content is uploaded to the KV store before updating metadata, and the order is reversed for deletion. If metadata failed to be updated, we will re-try processing the request for *max_attempt* times (currently set to 20) before aborting. In case of file uploads, changes to KV store will be reverted when requests failed by issuing a DELE request. For deletions, failing to update metadata would not revert the changes in KV store, with the reasoning that the file content was meant to be erased by the user anyways.

3.5 Webmail service

The frontend server allows users to view their email inboxes and send/delete/reply/forward their emails. All emails of a user are recorded in a mbox file as that of HW2, and we assign every existing email an ID for deletion. For sending emails, there are three cases that we discuss respectively.

- **From local to local:** To send local emails, the frontend server fetches the mbox file of the receiver from backend servers, appends the mail to mbox, and writes the mbox file back. To avoid conflicts that multiple frontend servers may write to the same mbox file, we use CPUT to guarantee sequential consistency.
- **From remote to local:** To accept emails from outside the system, we reuse the SMTP server in HW2. Specifically, there is a process (SMTP server) that is only responsible for receiving emails from remote users. Different from that of HW2, the SMTP server should write the mbox file to backend servers.
- **From local to remote:** To send emails to remote users, the frontend server builds an SMTP client according to the recommendation of the handout. Note that the frontend server uses the pku.edu.cn domain instead of localhost to avoid spam filtering, and it can only send to remote users (*e.g.*, seas.upenn.edu) with the VPN on.

4 BACKEND DESIGN

This section elaborates on the design and implementation details for backend servers. We describe how backend servers interact with other components (*e.g.*, master, frontend servers, and cluster group if node is primary) (§4.2), operations provided (§4.1), consistency, and fault-tolerance strategies (§4.3, §4.4, §4.5).

4.1 Operations provided

Operations provided by the backend servers enable: (1) frontend servers to interact with the KV-store; (2) communication with the master node to update cluster state (*e.g.*, node down/restart); (3) Primary and secondary sync-up during node initialization or after restart. Table (§2) is a summary of all backend operations, and details are discussed in the following subsections.

4.2 Interaction with others

When a backend server receives a request string, it will parse the string to a kv_command, and sends a string serialized from the kv_ret it generated as its result to other severs using Protobuf [3]. All information about the response such as status and value is stored in kv_ret, and all information such as command, key, and value is stored in kv_command it receives.

Master: When a backend server is up, it will first communicate with the master sending a BACKEND_INITIAL request string serialized from MasterRequest class to the master server. After receiving a FINISHED response from the master, it will create a new thread sending HEARTBEAT to the master and start to process other commands.

The backend server will receive CLUSTER command from the master server indicating the cluster group information whenever there's an update, where the first one is identified as the primary

Send from	Name	Forwarded & Logging	What the backend server will do
Master	CLUSTER	×	Update its cluster information (*e.g.*, which is primary)
	RESTART	×	Reinitialize the server and start processing
	KILL	×	Stops processing any incoming requests
	ALL	×	Send all key-value pairs of a user
Frontend	GETS	×	Get the value of a key
	PUTS	✓	Put a key-value pair
	CPUT	✓	Update the value if the key is matched
	DELE	✓	Delete a key-value pair
Backend	CKPT	✓	Do checkpointing
	SYNC	×	Sync its states with primary (for recovery)

Table 2: Operations provided in backend servers.

server in this group. The backend server will parse this kv_command and update correspondingly.

The backend server will perform corresponding operations that master sends such as KILL and RESTART.

Frontend: Only the primary backend servers will communicate with the frontend servers. As described in (§3.1), when one backend server receive (ALL, GETS, PUTS, CPUT, DELE) commands, it will perform corresponding operations to storage and sends back response string using serialized kv_ret when necessary (ALL, GETS). If it's a primary backend server, it will first foward the command it receives to the secondaries in its group.

Cluster group: For primary backend nodes, they will forward the same serialized kv_command string they received from the Frontend to the secondaries in their group.

After a backend server receive a RESTART command, they will send a SYNC request to the primary backend server for recovery. The information of the backend server will given by the master whenever there is an update on the primary. Whenever a primary backend receives a SYNC command, it will share its data stored with the sender (secondary).

4.3 Replication and Consistency

Our current setup includes nine backend nodes partitioned into three clusters. The master node designates one node as the primary in each cluster, and all other will become secondary. When receiving frontend server requests, the primary node is responsible for transmitting any requests that involve state updates (PUT, CPUT, DELE) and checkpointing (CKPT). KILL, RESTART, SYNC and other requests that require only read-access (GET, ALL) to the KV-store would not be forwarded. The primary is responsible for ordering the incoming KV-store related requests and assign each with a unique, monotonically increase sequence ID before forwarding, to ensure consistency. It will also assign each request with a monotonically increasing ID, which is used for checkpointing and recovery as elaborated in later sections.

4.4 Logging and Checkpointing

Update operations to the KV-store (PUT, CPUT, DELE) are cached for better performance, and will only be written back to disk during checkpointing, which is invoked through CKPT request every few seconds. Each backend node keeps a logging file that records the operation received, which will be replayed to restore in-memory state in case node crashes before the next checkpoint. The logging file also tracks which operations are included in the last checkpoint. This can be done by recording a last checkpoint ID, which is the sequence ID of the last operation performed before initiating each checkpoint. During recovery and syncing, this sequence ID recorded will be used to determine if it is necessary for the node to request all KV-store information from the primary.

4.5 Recovery and Synchronization

When a node is first initialized or restarting after killed/crash, it should setup cache if it is primary, or communicate with the primary to sync up if it is a secondary node. To learn its role (primary/secondary) and obtain primary's address information if applicable, the node will first inform the master node of its own address during initialization to indicate it is alive (with a BACKEND_INITIAL request), start to listen for connections and handle upcoming requests, and wait for master node's response before syncing. When receiving a BACKEND_INITIAL request, the master will broadcast the address information of the primary and all other alive nodes in the cluster. Note that after the broadcast, if the node is secondary, it might start receiving requests from the primary. However, as the secondary node could be out-of-sync, and the requests received could failed to be processed. However, this is accepted, as secondary will sync-up with primary later, which will overwrite secondary's in-memory and on-disk state.

If the newly initialized/restarted node is primary, this indicates that it is the only currently alive node in the cluster, then it will simply set up local logging file and in-memory cache, and start processing requests. If the node is secondary, then it will connect with the primary based on information received from the master node, and start syncing with the primary.

The syncing process involve several rounds of communication between the primary and secondary node, during which time the primary node will temporarily block from processing any upcoming requests or initiating new checkpoints, until a message from the secondary node indicating end-of-sync is received. To initialize the process, the secondary node will first send a SYNC message to primary and wait for response. When primary receives a SYNC request, it will send over its local logging file, which will be used by the secondary to determine whether a full sync-up with the primary is needed, in which case the primary will send all on-disk checkpoints and logging to the secondary. As discussed in previous sections, the logging file also keeps track of the last checkpoint ID, which equals the sequence ID of the last operation included in the latest checkpoint. For instance, suppose during the syncing process, primary has just finished processing a request with sequence ID 15, and also checkpointed, then 15 will be recorded as the last operation included in this checkpoint. When secondary receives the logging file from primary, it will read the last checkpoint ID included in the primary file, and compare it with the last checkpoint ID in its local logging file. If secondary's local last checkpoint ID is equal to primary (it could not be larger than primary assuming all operations and CKPT requests are properly forwarded by primary, and that secondary cannot initiate checkpoint process by itself), then it means the primary has not yet checkpointed while the secondary is down. The secondary can simply overwrites its local logging file with the content received from primary, clear up in-memory caches, and replay to recovery in-memory state to sync-up. When finished, the secondary will first send a OK request to indicate no FULL sync is needed, and a SYNC_DONE message to primary to inform the ending of syncing process.

If the primary's last checkpoint ID is larger than that of the secondary, then a full sync-up will be requested, as the primary must have checkpointed before secondary starts, and the KV-store state on-disk could potentially be out-of-sync. To request a full sync-up, the secondary will send a FULL_SYNC message to primary, clear up all on-disk checkpoints, logging, and in-memory caches, and waiting for all information to be transmitted by the primary. Upon receiving this request, primary will transmit all files on-disk associate with the KV-store to secondary, including the logging file, and wait for a response sent by secondary after syncing finishes. The secondary will use the files received to recover its on-disk and in-memory state, and inform the primary when after received all files and replayed all logging, which will effectively end the syncing process. Both primary and the newly started secondary will then start processing requests. If any critical failures happen during the syncing process (*e.g.*, failed to transmit specific files), error will be reported, and the secondary node will abort initiation and exit.

5 CONTRIBUTION

The contribution of the team members in this project is as follows

- **Yichun Cao:** Webmail service User Interface, Frontend server, Admin, Master node, Report composition
- **Chen Fan:** Main pages and Storage service User Interface, Frontend server, Backend server, Inter-server communication, Report composition
- **Yazhe Feng:** Storage service, Consistency, Logging & Checkpointing, Recovery & Syncing, Backend server, Inter-server communication, Report composition
- **Yinda Zhang:** Load balance, Monitoring, Master node, User authentication, Webmail service, Backend server, Inter-server communication, Checkpointing, Report composition

REFERENCES

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[2] Roy Fielding and Julian Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. Technical report, 2014.

[3] Kenton Varda. Protocol buffers. http://code.google.com/apis/protocolbuffers/.