CIS 505 Final Project Report

Reno Kriz, Hari Krishnan Ramachandran,Vikram Nag Ashoka, Selina Hsu-Ying Liu

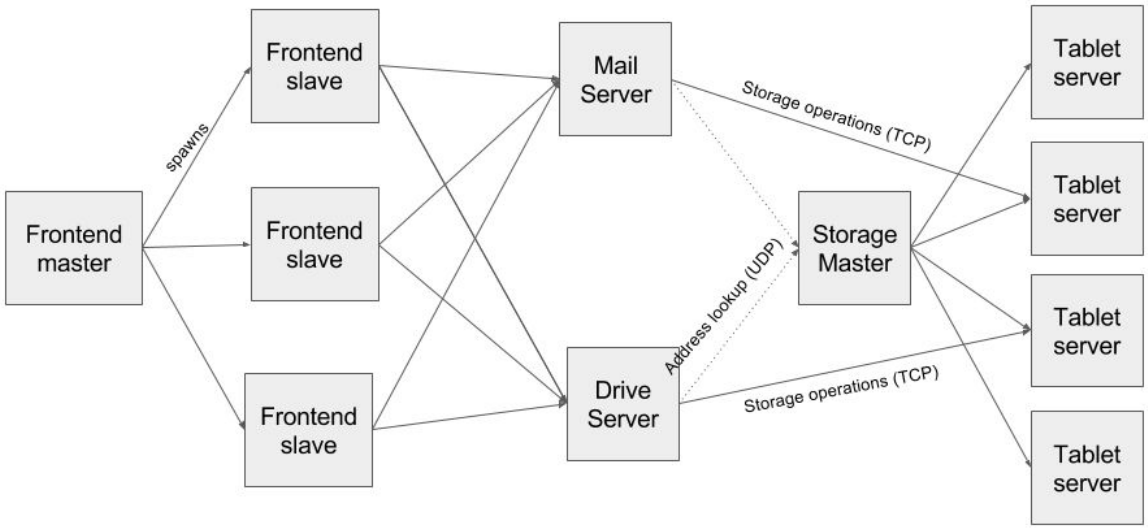

Diagram 1: architecture of our PennCloud

1. Key-Value Store

The key-value store is made up of a single master node and a cluster of tablet servers. The tablet servers maintain partitions of the table based on key ranges, while the master server maintains membership information of the cluster and directs clients to the appropriate tablet server based on the row key queried.

Exposed API
Initially, the client has to contact the master server via UDP for the appropriate tablet server to connect to for a specific row.

Client: query linh
Master: 127.0.0.1:5001

Client will then set up a TCP connection with the tablet server node identified by the address “127.0.01:5001”, and start a client session with “yo\r\n”:

Client: yo\r\n
Server: +OK hi
Client: put linh,mail 5 1
Server: +OK go ahead

Client: <blob>

Replication
The cluster adopts the primary/backup approach. The replication scheme can be configured in the configuration file. In the current config file, the table is split by row key ranges into three tablet servers, replicated on two servers each, one of which is the primary. Initially, the first tablet server listed in each key range sub-cluster is set as the primary.

Fault tolerance
The master server keeps track of the health of the cluster by maintaining a TCP connection and sending “ping” regularly to each tablet server. Each TCP connection is started and managed on a new thread. If a tablet server does not respond with “pong” within a set time limit (say 5 seconds), the master server will declare the server crashed and send “crash <address:port of crashed server>” to the rest of the servers in the same key range sub-cluster, for them to update membership information.

If the crashed tablet server happened to be a primary server, the master server will select the next healthy tablet server responsible for the same key range and elect it to become the new primary by sending “elect <address:port of new server>” to the rest of backup servers, who would respond with “+ACK”.

After declaring server crash (and electing a new primary if need be), the master server will try to restart the crashed node by forking a process to execute a new instance of the tablet server, through invoking “./tabletvm -r config.txt <index>” to the system. After the server has successfully restarted and recovered (through the mechanism detailed in the next section), the master server will send it “elect <address:port of new primary server>” to inform it about the new primary server.

Logging and recovery
Each tablet server keeps a log file on disk and logs a record for every write operation (put/delete) to the underlying tablet. The tablet server also takes a checkpoint every other minute, by serializing the underlying map and writing it to a new checkpoint file on disk with the help of Boost Serialization. To free up disk space, the server will also clear the log and delete the third last checkpoint file after writing a new checkpoint file.

When a tablet server is recovered after a crash - by the master server invoking “./tabletvm -v -r config.txt <index>” on the command line with “-r” flag - the restarted server will deserialize the latest checkpoint file in the local system to reconstruct the tablet, and then replay the corresponding log file (identified by key range and server address/port combination) to the reconstructed tablet to get to the state right before the crash.

Running the service

1. Create ./logs and ./snapshots subdirectories in the storage directory. Start the tablet servers first.
2. For the ith tabet server (starting from 0 to n-1), run:
   ./tabletvm -v config.txt i

3. After starting all the tablet servers, run the master server:
   ./mastervm -v config.txt

Challenges
One key challenge durin implementation is the fact that the OS takes a while to release addresses bound to a socket, even after a program terminates. This means that the master server has to wait 10-20 seconds before restarting a crashed server using the same address, or else the restarted process will terminate again.

2. Webmail Service

We implemented a basic mail server that interacted with both the front end server, as well as the key-value storage servers. This mail server supports five main commands. The first was GET_INBOX, which returns the list of emails sent to a user; an example of this command would look like “GET_INBOX~<username>\r\n”. The second command is GET_SENT, which returns the list of emails a user has sent; an example of this command would look like “GET_SENT~<username>\r\n”. The third command is “GET_EMAIL, which returns the sender, receiver, header, and body of a specific email; an example of this command would look like “GET_EMAIL~<username>~<header>\r\n”. The fourth command is SEND_EMAIL, which returns a +OK if the email is sent correctly; an example of this command would look like “SEND_EMAIL~<sender>~<receiver>~<header>~<text>”. Finally, the last command is QUIT, which closes out of the webmail server.

When initialized for a new user, the frontend server creates a new row in the key-value storage server by calling the master node. However, it does not create any columns for the mail server. Thus, when the frontend sends one of the first four commands (excluding QUIT), the mail server first sends a UDP package to the master node of the key-value storage server to get the row of the current user, and the port of the appropriate tablet to use. From there, the mail server sends a GET request to tablet using a TCP connection to check if the inbox of that user has been initialized. If not, the mail server sends a PUT request to put a “.” into both the INBOX and the SENT column, to represent an empty list. After checking this, the mail server sends the respective request to the key-value storage server. For each subsequent request, if the current user remains the same, the mail server does not check the inbox, because it already knows that this has been initialized. Otherwise, if the current user changes, the mail server requests the master node for access to the row of the new user, and checks if the inbox has been initialized, before continuing to the command.

When sending an email, the mail server first sends a GET request to get the list of sent emails from the cell (<sender>, <SENT>). From here, the mail server sends a CPUT request to replace the old list of sent emails with a new list, which appends the newest email header to the top of the list. From here, the mail server sends a PUT request to put the full email in the cell

(<sender>, <header>). The mail server then does the same for the receiver, replacing SENT with INBOX. All data is first converted to a binary string before being sent to the key-value storage server, and converted back to a string before being returned to the frontend server.

During implementation, we faced several challenges. One issue was how to deal with connections to several sockets at the same time, as the mail server needed to connect to the frontend server, as well as the master node and tablets of the key-value storage server. We originally thought to use select to listen to both the frontend and the key-value storage server, but then realized we did not need to listen for anything incoming from the backend. Additional features we attempted to implement included folder functionality, as well as the ability to send/receive to emails outside of our system, but due to limited time and issues with integrating everyone's code together, we decided to only include the features that we knew would work for the final submission. If we were to redo this part, one major thing we would do is have the storage server and webmail server commands formatted the same way, to make it easier for the frontend to communicate with both without using different sets of commands.

3. Frontend Server

High-level design
The front-end server is made up of a single master node and a cluster of several subordinate servers which are spawned by the master during runtime. Master node redirects incoming clients to one of the subordinates using load balancing techniques. Master node also supports an admin console.

Supported Services
Whenever a client browser request reaches the Master node, it redirects the request to the subordinate with the least load. The client then handles all the subsequent requests from the client.
When the client Logs in, the frontend server sends a GET request to the storage server to obtain the username's corresponding password. It then compares this record with the password entered by the user. If successful, the user sets a cookie to the client browser to keep track of user session (this cookie is cleared when the user logs out). The frontend server then contacts the webmail server or the drive server for corresponding user requests.

Admin console
The admin console is a special functionality provided only by the Master node. When accessed, it displays the list of all subordinate frontend servers, and their states (alive or dead). Along with this data, it also displays a list of storage servers and their states, and also the key value pairs stored in all of them.

4. Drive service

We implemented a drive service using which users were able to upload and download files. Other functionalities included, creating a folder, moving files into a folder, deleting a file or a folder. Drive service was also used to keep track of directory structure for each user.
Here the drive was connected to the front end service and the key value store. Drive received commands from front end server which included username and the path to a file or a folder. Depending on the user's action information received from the front end server varied from only username and path to a file for upload/download to username , old path and new path for a case of moving a file. To keep track of various folders and files in each user's drive, we created files where we wrote information about directory structure for a particular user.

For the case of uploading a file, we first the read the file which was opened in binary mode in chunks of 256 bytes. Then the chunks were all appended together and stored in key value store.After this there was an entry made into the file which was used to keep track of the directory structure of the user so that soon after he makes an upload he will be able to see the file he uploaded. For the case of downloading a file, it was first fetched from the key value store. Then this was transferred to the front end server in chunks of 256 bytes. The front end server again appended all these chunks and wrote them to a file with the same name as the user specified one and sent it to user. We support both upload and download for text files, images of all formats.

Some of the commands supported by drive:

#Contents# -> To obtain contents of the folder of a particular user.

#Upload# / #Download# -> To upload or download a file.Upload triggered a dialog box where user could select a file. Download was triggered when user clicked on a file.

#DeleteFile# / #DeleteFolder# -> when user selects the delete option for a file or folder.
Major challenges were for implementing uploading and downloading a file where we tried many different ways such as using the boost Serialization through which we can read data from user in a specified format. But then this would be proper only if we were supporting files of a single format. So we decided on reading in binary format which would eliminate all the format constraints.

For other cases such as deletion and moving, first there were changes made to the file which was used to keep track of the directory structure. Then on the backend, files or folders which were moved were written to their new paths in the key value store. Also deleting the old values in the key value store. If we had to redo this, then it would be better to integrate drive and mail more concisely so that there is less communication with frontend.