PennCloud - CIS 505 Software Systems Final Project Report Fall 2020

Jayanth Bhargav, Nagarakshith M.S., Vikas Shankarathota, Vishrutha Konappa Reddy

School of Engineering and Applied Sciences, University of Pennsylvania

Abstract

This report describes the design of a fully distributed cloud service supporting web-mail and storage services. Section 1 describes a brief overview of the architecture of the cloud service and the various components and how they interact. Section 2 describes the key-features developed/implemented in this cloud service. Section 3 and Section 5 describe in-detail the various components, design choices and challenges in the Front-end and Back-end systems respectively. Section 4 describes the inter-process communication setup used in the system. In Section 6, the contributions of the each of the team members with respect to various components of the project is listed.

1 Architecture

The cloud service is built of a large number of servers running together in a distributed setting. There are two major parts: Front-end and Back-end. The Front-end consists of a Load Balancer/ Master server which accepts client requests and redirects them to one of the front-end HTTP servers which serve the client. There are 3 HTTP servers. These servers interact with both the client and then back-end servers to serve the client requests. Both the master servers monitor the status of the slave servers which send their heartbeat messages at regular intervals.

The back-end server stores all the data of the clients and is a Key-Value store which is analogous to the BigTable. The Key-value stores are distributed over 3 different servers and a Master Server maintains the information of the distribution of data.
There are 3 SMTP servers which mail relay to local and non-local users. An admin console, which can be accessed with a special address is present which shows the status of all nodes in the system and and can kill or restart a node during run time. Figure 1 depicts the architecture diagram of the system.

2 Key Features

The key features which the cloud service provides are:

1. User Accounts - New user sign up, Login authentication of user
2. Front-end server - Handles GET/ POST/ etc.. from clients, handles Cookies/Sessions for clients
3. Webmail Service - Local and non-local mail relay, Inbox service to view and delete emails
4. Drive storage - Create, Move, Rename and Delete for Folders and Simple files types (large file sizes also supported)
5. Key-value store - Supports GET, PUT, MOVE, RENAME, and DELETE operations. Additionally - Primary-based Replication (2 Phase Commit), Logging, Checkpointing and Recovery as part of Fault Tolerance.
6. Admin Console - A special server which displays the status (alive/dead) of all nodes in the system and can kill/restart nodes during run time

3 Frontend System Design Description

This section describes the design of the key components and sub-systems in the front-end of the cloud service.

3.1 Front-end server

The front-end server consists of a main HTTP server (master-server) which handles requests from the clients and redirects the client to one of the slaves. Clients will only need to know the address of this master web server. The master front-end has the following functions:

- Issue permanent redirects to client requests to one of the active front end HTTP servers
- Ensure load balancing at the front-end
- Monitor the status of HTTP and SMTP servers (alive/dead) and report to admin console

There are three HTTP servers to serve client requests. The HTTP servers are built in compliance with RFC 2616 [Henrik Nielsen(1999)]. Once a client gets redirect from the load balancer to one of these servers, these servers will start processing the requests and interacting with clients. The HTTP servers have the following functionality:

- Render HTML/CSS/JS pages to the client

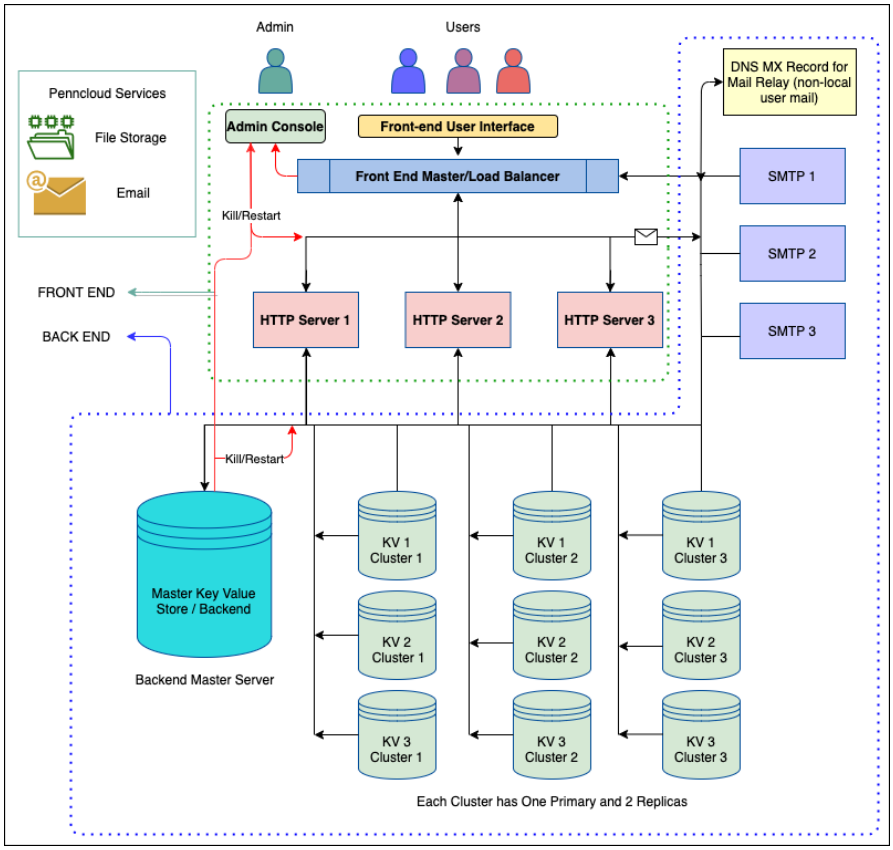

Figure 1: Architecture Diagram

- Manage cookies/sessions and replicate them at the back-end and recover them in case of frontend server failure
- Handle GET/POST requests (GET for resources which the client requests resources and POST for information collected from the client)
- Handle client authentication and authorization by communicating with backend servers
- Communicate with back end for files/folders/resources asked by the client
- Fetch the emails stored at the Key-value store and display the inbox for the client

3.2 Front-end Rendering

The front end rendering uses HTML, CSS and Javascript in order to display dynamic data to the user. Major functionalities that the user is able to use include: sign up page, login page, admin console, email service (sending and inbox), and a file storage system to upload, download, and view simple files.

3.3 Front-end Replication and Fault Tolerance

If one of the HTTP servers fail/crash, a client who is interacting with that server will receive a connection reset/ unable to connect to Host message in the browser. In such a case, the client can just head back to the front-end load balancer/master, which is redirect it back to one of the active HTTP front end node and this node will not ask for user authentication and just have the user already logged in. This is achieved by replicating the Cookie/Session ID information of the client at the back-end. When another nodes starts receiving queries from the client, it in turn queries the back end for the session information and will not ask the client to log in again if it was previously authorised and an active session exists. In essence, the session of the client gets replicated onto the new front-end server.

3.4 Email Server

A cluster of multi-threaded SMTP servers will relay emails from users to the key-value store in case of local users and will relay the mail to non-local users by looking up MX records in the DNS. The front-end HTTP servers fetch the stored emails from the key-value store to the user. The client only interacts with the front-end HTTP server for all email transactions. The SMTP server is built in compliance with RFC 821 [Postel(1982)]. For accessing emails (viewing, deleting) which are delivered to a client, the HTTP servers directly queries the Key-value store and performs GET/DELETE operations on the same.

3.5 Admin Console

The admin console is a special server (HTTP server) through which a user having admin access can login in and view all the nodes of the system. The admin console provides the following features:

- View the status (ALIVE/DEAD) of all the nodes in the system
- Kill and Restart nodes during run-time (which is useful for testing replication/logging/recovery of the nodes)

The admin server communicates with the front-end master and KV master servers to query for the status of all the nodes in the system.

3.6 Design Choices and Obstacles

All the front-end HTTP servers and SMTP servers are multi-threaded. A thread-pool having a Job-Queue is implemented which has a configurable number of worker threads and queue size. This is a highly scalable solution as it can handle a lot of client requests without having the overheads of creating and exiting from new threads. The thread pool will have N threads always serving a job and once it finished a job, it will dequeue from the job queue. Another major design choice was to use a minimal proto library, such as Google Protobuf, which includes writing out all the server, sockets, event handlers and message creation functions. This is a substantial amount of work compared to using serialization packages like gRPC in which many operations like starting a server and reading incoming messages is already well abstracted.

4 Inter-Process Communication

For inter-process communication, the system uses Google's Protocol Buffer [Varda(2008)]. Google's protobuf allowed us to serialize and deserialize communication over sockets and read out different parts of the message without having to worry much about the marshalling steps. In order to integrate protobuf well into our architecture we wrote wrappers that easily allowed us to create the required message on the sender's side and after deserializing the message on the receiver side, we were able to obtain the message type. The system consisted of around 15 different types of proto messages.

5 Back-end System Design Description

5.1 Key-Value Store

The key-value store is the backend datastore on which all the user data and metadata is stored in a distributed manner. The KV store is implemented to emulate(in simple terms) Google's BigTable [Chang(2008)]. The key and the value are both of *string* datatype and the tables are sparse. The KV store consists of one KV Master node and several clusters with each consisting of a primary node and configurable number of secondary/replica nodes. The KV Master node performs the following operations:

- Performs load balancing by assigning clusters according to information queried.
- Assigns a new primary amongst the active replicas in round robin fashion when current primary dies.
- Keeps track of alive nodes in a cluster by monitoring a heartbeat message.

It is assumed that there is always one active node in a cluster at all times. The distributed nature of KV is ensured by sharding the tables(into tablets) according to username. The project makes use of three tables, one for metadata, one for emails and one for files and folders, where the rows correspond to a particular user and the columns are particular fields of information. Five operations:

- PUT
- GET
- DELETE
- MOVE
- RENAME

provide the means for the HTTP and SMTP servers to insert and manipulate data in the KV store. Other than the above five operations, a cluster node also sends a timed heartbeat message to keep the Master node updated about it's status.

5.2 Replication and Consistency

To ensure that the Key-Value store is resilient to failures, data is replicated in all the nodes of a particular cluster. *Primary-based Replication* is the protocol used to do this. With replication, consistency of data is important and to ensure this *Sequential Consistency* along with a modified version of the 2-PC protocol was implemented. When a primary receives any one of the API requests listed above, it issues the PREPARE command to all it's secondary nodes. The primary then sends the COMMIT message to all the active nodes. This ensures that for a particular row(user) all operations and forwarding happens in identical order across all nodes. After sending the COMMIT message, the primary performs the API operation locally.

5.3 Logging and Checkpointing

Every time any node receives an API request to perform any operation locally, the node immediately force writes the operation and the details of the call onto the local log file. It then performs the respective actions, such as fetching existing values from cache, putting a value, modifying column values on the in memory cache. The log file and the cache will later be accessed during checkpointing. The KV store is programmed to checkpoint at regular intervals, and the primary performs and sends out a command to all its replicas to begin checkpointing. The checkpointing process involves two steps:

- Migrating data from in memory to disc.
- Creating a checkpoint file for version control during recovery.

Migration is done by consulting and performing operations sequentially as it occurs in the log file, fetching appropriate data from cache and inserting it into the disc. The log messages are also used to create and keep track of a checkpoint file, labeled as *checkpoint_VN.txt*, where *VN* is version number, which contains the version number of a particular file or directory structure. This file is updated at every checkpoint and is used during the recovery process by the node. Once all the commits in the log file is processed, and the checkpoint file is updated, the log file is cleared.

5.4 Recovery

The recovery process begins immediately after a node is revived. The node first contacts the KV Master to learn which of the nodes in the cluster is the primary. The revived node then uses this information to contact the primary and learn the latest checkpoint version. It then compares it's own checkpoint version with the primary's. If the version numbers are different, the secondary node requests for the latest checkpoint file and compares the version numbers of existing files and folders or the presence of any new components that needs to be created. All the neccessary operations, such as creating, renaming, moving or deleting of files and folders are performed on the disc. Once the two checkpoint files sync, the on disc structure is translated on to memory cache. The next step is to obtain the log file from the primary. Once the log file is received, the file is read sequentially and the node's cache is brought up to date by performing the operations. After performing these two steps, the disc, checkpoint file, and the log file of the revived node is synced and the recovery is done. During this process the primary server will not handle any requests from the frontend.

5.5 Design Choices and Obstacles

As part of the consistency process, the 2-PC protocol was modified to suit the needs of the project. Traditionally when the primary node does not receive a reply to the PREPARE message from any one of the secondary node, it is blocked until it receives a confirmation from all. The modified 2-PC was implemented omitting this step. The primary sends out the COMMIT to all it's active secondary nodes. This was done so as to keep the flow going. The dead node on revival performs self recovery and updates itself.
One of the obstacles encountered was due to the choice of keeping the key and value of the KV store to be of string type. This hindered the upload of binary files such as images, GIFs, etc. Since images are uploaded are binary array, conversion of binary to string during upload and conversion back to binary during download is lossy and leads to erroneous results. This lead to the project not being able to support images, GIFs and other binary files. However, the cloud service is capable of handling uploads and downloads of large file sizes.

6 Contributions

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

- Jayanth Bhargav - HTTP server, Front-end rendering, Front-end Replication/Fault Tolerance, API's for Back-end and SMTP query, HTML/CSS for Front-end
- Nagarakshith M.S. - SMTP server, Backend Replication, API calls for Back-end query, Master Nodes, ThreadPool Wrapper Classes
- Vikas Shankarathota - Admin Console, Frontend JS/HTML for rendering, Inter-server communication API (Built on google::protobuf)
- Vishrutha Konappa Reddy - Key value store, Backend Replication, Logging, Checkpointing and Recovery.

References

[Henrik Nielsen(1999)] Larry M Masinter Roy T. Fielding Jim Gettys Paul J. Leach Tim Berners-Lee Henrik Nielsen, Jeffrey Mogul. Hypertext Transfer Protocol – HTTP/1.1. (2616), June 1999. doi: 10.17487/RFC2616. URL `https://rfc-editor.org/rfc/rfc2616.txt`.

[Postel(1982)] Protoco SMLT Postel, Jonathan B. Rfc 821. *Simple Mail Transfer Protocol [online]*, 1982.

[Varda(2008)] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72, 2008.

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