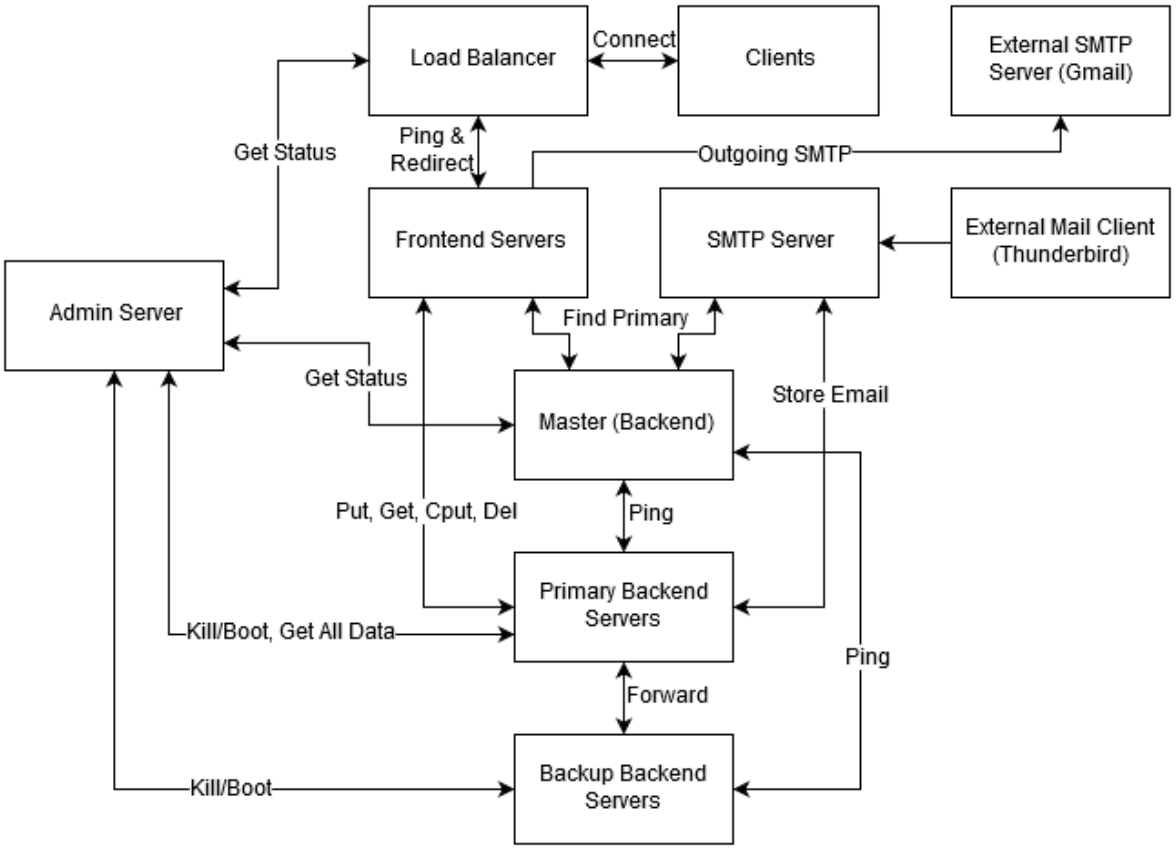
PennCloud Report: Team 04

Xuerui Fa, Emmett Neyman, Jason Schwartz, Jared Winograd

I. Design Overview



The figure above displays the components of our system, how they interact with each other, and what information is requested/sent in each interaction. Our implementation uses protobuf messages internally between nodes.

A sample workflow of our program could look like this: a client (firefox) connects to the load balancer, which redirects the client to one of the frontend servers. The frontend server, which was launched knowing the IP and port of the Master backend server, connects to the Master server to receive the IP and port of the primary backend server that it needs. The Master node frequently pings all primary/backup servers to see if the node is up or down. The primary and backend servers are responsible for maintaining the file store, with primary nodes forwarding requests to its corresponding backup servers. The frontend server, once it has connected to the appropriate primary, is able to perform put/get/cput/del operations.

The frontend server can also communicate with external SMTP servers, such as gmail, to send a new email. Additionally, our SMTP server can receive SMTP from email clients such as Thunderbird, and store these emails in our file store in the same way in which the frontend servers communicate with primary storage nodes.

Finally, our admin server provides access to the status of servers on our network. The admin node requests status messages from load balancer and Master- these nodes know the up/down status of the frontend/backend storage nodes, respectively. The admin server then displays this information in a webpage. It also provides the ability to kill/reboot backend servers (primary and backup) via html buttons, and it displays all the columns/values currently stored in the filesystem. It does this by contacting the primary backend servers to get a list of the current values in the filestore from each one.

II. Features Implemented

Frontend (Jared, Emmett)
- Identify user based on cookies
- Load-balancing to frontend servers

Email (Jared, Emmett, Jason)
- Viewing emails in user's inbox and opening individual emails
- Deleting emails, replying to emails, and forwarding emails
- Sending emails to an external email address (emmettneyman@gmail.com for example)
- Receiving emails from local SMTP clients (Thunderbird for example)

File Storage (Jared, Emmett)
- Uploading new files of any filetype
- Creating new folders
- Renaming files and folders
- Moving files and folders into and out of folders
- Downloading files

Admin Console (Emmett, Xuerui, Jason)
- View information about which frontend servers are up
- View information about which backend servers are up
- Killing and rebooting individual backend servers
- View the contents of each of the individual tablets

Backend Storage (Xuerui, Jason)
- Primary replication
- Checkpointing
- Fault tolerance (recovery after crash)
- User folder locks

III. Design Decisions & Major Challenges

A. Frontend

There were two major design decision for the frontend servers. The first was deciding how the load balancer should distribute new connections to the various frontend servers. The second was deciding on the schema to use for storing emails and files in the backend key-value store.

Load Balancing

The job of the load balancer node is twofold. First, the load balancer must handle all incoming HTTP connections to our service and redirect the clients to different frontend servers in a manner that evenly distributes load. The second job is to communicate with the admin console server about which frontend nodes are alive and which are down. Everytime the load balancer receives a new connection, it sends a PING to each of the frontend servers and waits to hear get an ACK message back from each server. The ACK it receives contains information on that server's load. Specifically each server tells the load balancer how many HTTP requests it has already processed. If the connection to the load balancer is a GET request, the load balancer will pick the server with the smallest load and redirect the client to that node. If the connection to the load balancer is a PING from the admin console, the load balancer will send an ACK message back to the admin console containing information about which frontend servers are up or down (it gets this information from its initial PINGing of each of the frontend servers).

File/Email Storage Schema

When a user uploads a file or creates a folder, we generate a distinct node (a "node" is either a file or a folder) ID by combining the user's name, the node's name, and a unique 4-digit number. This is so that files with the same name can be distinguished from each other. In the key-value storage, our rows are the users' names. We then store the following columns for each node:

- **[nodeID]:name** - contains the node's display name (e.g. "myfile1" or "myfolder1")
- **[nodeID]:content** - contains the node's contents. For files, this is the actual file contents encoded in base64. For folders, this is the list of nodeIDs of the nodes that are inside the folder. This allows us to find and display nodes by recursively searching through the nodes' contents.
- **[nodeID]:path** - contains the node's path (e.g. "/myfolder1/myfolder2/myfile1"). This allows us to find a file by its path, for purposes of renaming, moving and deleting.
- **[nodeID]:parentID** - contains the ID of the node's parent. For nodes in the top-level directory, this is the word "root". This allows us to remove a node from a parent's records easily.

When a user sends an email to a local user, we generate a distinct email ID by combining the email's arrival time with a unique 4-digit number. This is so that emails with the same subject and contents can be differentiated from each other. In the key-value storage, our rows are the users' names, just like for files. We then store the following columns for each email:

- **[emailID]:contents** - contains the email's contents
- **[emailID]:arrivalTime** - contains the email's arrival time.
- **[emailID]:sender** - contains the email sender's name
- **[emailID]:subject** - contains the email's subject

B. Backend

Primary Replication

We decided to use primary-based replication for backups. We chose primary replication because it abstracts away a large portion of how storage works for the frontend servers. All the frontend needs to know is the primary server responsible for the tablet that it wants to store into.

If the frontend isn't able to connect to a primary, it asks the master node what the current primary for that tablet is. Once the master responds, the frontend server will then proceed to connect with that primary. Thus, the frontend only ever needs to know one server for each tablet, keeping overhead for communication between frontend and data storage minimal.

Finally, primary replication is simpler than quorum replication to implement, as all writes to the replicas are forwarded from the primary. In addition, only the primary needs to handle GET requests from the data storage.

Fault Tolerance
When a node comes back online after it crashes, the master backend server detects this and informs the current primary for that tablet. The primary then transfers the most recent checkpoint to the newly booted node, thus maintaining the invariant that all primary and backup servers for a given tablet store the same data.

Sequential Consistency
Our system uses primary replication, and as a result, we are able to achieve sequential consistency throughout our storage nodes. The primary node for a tablet handles requests as they are sent in, and with our system of user locks, a specific user will be modified in the order that these requests come in. In addition, these requests are sent to the replica nodes in the order they are processed on the primary, ensuring that the replicas will handle the requests in the same order as the primary.

Locks on User Folders
When implementing locks, we decided to create mutex locks on the user folder level, as opposed to on the tablet level or the user column level. Locks on modifying the filesystem tablets would be too strict, as only one thread on the backend server would be able to insert, fetch, or delete data from the tablet at a time. This would defeat the purpose of having multi-threaded backend servers. On the other hand, mutex locks on user columns require a significant amount of overhead to handle. Mutex locks would have to be constantly created and destroyed, whenever new data is added or old data is deleted. In addition, when we need to prevent any further changes to any user on the tablet (in the case when we are executing a recovery or checkpoint), if locks are handled per user column, we would need to obtain the mutex lock for every column in every user. By implementing locks for every user, we are able to achieve a balance between simplicity and performance.

Protobufs
Our system uses protobufs to transmit information between server nodes. Protobufs is a useful library in that the methods for creating a data object, setting individual fields, and serializing/deserializing the object are automatically generated by the protobuf compiler. Thus, it's very straightforward and safe to send information as a protobuf object.