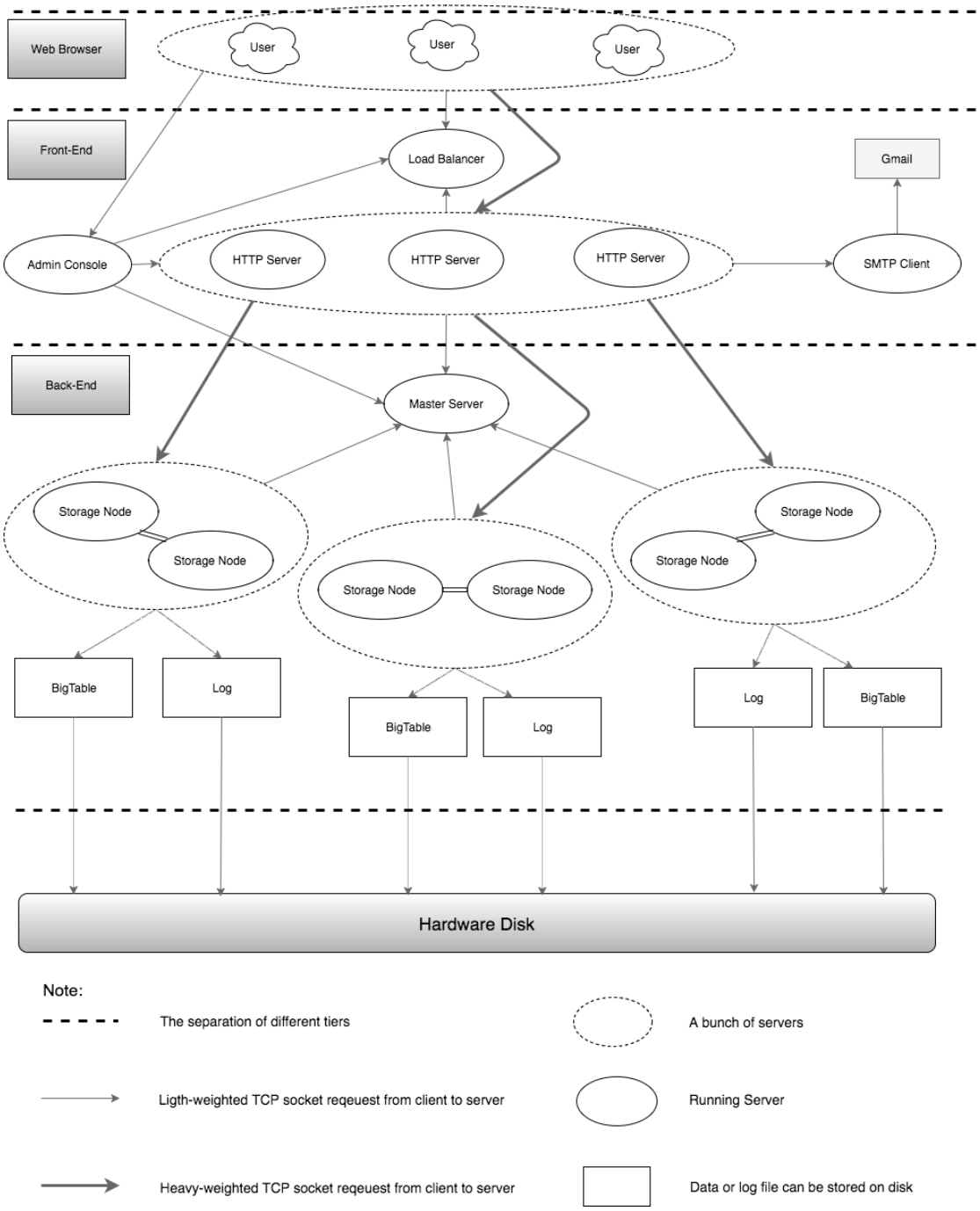
CIS 505 Final Project Report

Team Members: Jian Li, Yang Yu, Lijun Mao, Ziyu Chen

System Architecture



Features

Frontend

- Web server

General design:

Bridges users and backend server. After getting requests from users http servers communicate with Master to get the address the storage server that has data it wants and communicate with storage server to get, and put data.

Fully distributed. Multiple servers to handle possible crashing and load balancing. If one http server crashed, user requests will be redirected to another server.

Multithreaded and connectionless. Upon receiving a new request, a new thread would be created to handle and forward the request.

Use cookies to manage session.

User account:

Implemented login, signup and change password functions for user to interact with the site. Use cookie to manage user's information and achieve user sessions in a connectionless server. We implemented protocols LOGIN, LOGOUT, REG and PASS to communicate with backends to manage user data.

File drive:

One of the most important part of the http server is to provide a series of drive operations to achieve uploading, storing files into the cloud(our storage servers) as well as downloading files from the cloud. Specifically, it supports create, open, rename, remove operations on directory, and provides upload, download, move, rename and delete file manipulations. The protocols include GET, PUT, DEL, MOVE, VIEW and DIR, for example, every time user wants to upload a file, after acquiring the file data user send PUT along with user name, file relative address, file type and file content to backend, more details about the protocols described in 'protocols.txt'.

Upload and download data:

Parsing and encoding is also a critical part of the http server which allows different kinds of file can be transmitted through the system as binary data without data loss. During upload phase, the server parse binary data from uploaded file sent from browser and then encode data into hex string, and then send to backend server, the backend server stores the data without further modifying. During download phase, the http server acquire the hex string from backend storage and send the data to the browser.

Email interface:

The web server provided an email interface for users to send emails in our @penncloud webmail system or to the remote webmail such as Gmail. When composing a new email, the server has to tell if the receiver is our webmail user. If the receiver of an email

is in our webmail system, the web server will notify the master with information of the receiver and the title of the email, then read the storage node address and write the data to corresponding storage node. In spite of the primary storage node, the web server should also send a copy to its replica to ensure data consistency and fault tolerance in case of server failure. If the receiver is a Gmail user, web server will simply send the title and content to the smtp client for connection with remote webmail server. Users can view all their receiving emails in the inbox page. They can also operate on these emails such as reading, deleting, replying, or forwarding.

- Load balancer

A server between users and http servers for Initial filtering of requests and redirecting users to a web server, distribute workload on http servers using round robin algorithm. Keep track of and stores states(alive or down) of web servers, every time a http server is run, load balancer will receive a notification from that , similar when a server a down, load balancer will receive notification and update the states.
Finally load balancer need to listen and respond admin console with the states of web servers.

- Admin console

A small server with user interface that monitor the states of all servers including front end web servers and backend storage servers, updates the states upon refreshing. In the admin interface, it list all the servers with their ip and port, and their states.
Control servers shutdown: provides shutdown button to shutdown a currently running server, to finish this action, admin console will contact the target server directly and close them, and then update its own state stable.
Show raw data: admin console is able to display the raw data we stored in storage servers, each data displayed as an entry(user, filename, filetype, and the servers addresses they are stored in).

Backend

- Master Server

The master server works as the coordinator of the entire system. It is responsible for message routing, backend load balancing, backend status management and fault tolerance. The master is crucial for almost everything. It should never crash for functionality of entire system.
1, As a router
All frontend servers need to contact the master before they want to operate on bigtable. The master will select a proper backend server and give its address back to frontend. For read operation, the server which manages corresponding tablet will be selected. And for write operation, the server which has smallest load will be selected.

2, As a backend load balancer

We use number of connections and size of tablet to quantify the load of each backend server. The master monitors the load of each backend server and assigns server with smallest load to new frontend write operation. Whenever a backend server address is sent to frontend, master knows this backend server will be connected. Hence master increases the connection counter of this server. Whenever an operation finishes, the backend server will notify the master and master decreases the corresponding counter. But since we don't have stress test so far, the effect of load balancing is not clearly seen.

3, As a backend status manager

All backend servers have persistent connections with master when started. The master server monitors all backend servers and mark each one WORKING or CRASHED in a table. The master also assigns the role of each backend server dynamically. Any backend server can be PRIMARY or REPLICA in different situations.

4, As a fault handler

We have 2 backend servers in a group. From the perspective of quorum, R=1 and W=2. In other words, frontend servers write in both servers and only read from one server(PRIMARY). Whenever a backend server crashes, the master will notice it and mark the server as CRASHED. And the remaining server in the group will be assigned PRIMARY despite its original role. And when the crashed server resumes working, it becomes REPLICA despite its role before crashing.

The crashed server will load all the checkpoint information from disk after restarted. And it will connect to the master and ask for the address of the other server in the same group. The master will give the address back. And the server will contact this address and try to get logs for full recovery.

- Storage Server

The storage servers are responsible for user data storage and management. Each storage manages a tablet(part of bigtable) and perform corresponding read/write operations on it according to requests of frontend. The storage servers work in pairs for fault tolerance purpose. If one crashes, the other could continue working and help the crashed one to recover after restarted.

1, When Working

Each request to the server will be handled by a corresponding thread. The server will analyze the request and call related bigtable API(GET, PUT, DELETE, ETC) and admin API(SHUTDOWN).

2, When Crashed

The other server in the same group will continue to work. All servers maintain its own checkpoint and logs. For consistency, the time between crash and restart should be no

more than a checkpoint time(can be adjusted in codes). If both servers in the same group crash, most data will be lost.

3, When Restarted

The restarted server will first recover checkpoint from disk. Then it will ask master for the address of the other server in same group and ask that server for log. Finally it recovers full information from that log and starts to work as a replica.

- BigTable API

According to what we learned in class and BigTable paper, we design our storage node as a simplified BigTable. We use the username as the row key and encode the filename and the file type in the column key. And we design a struct File_Data to encapsulate all the metadata of a content that we want to store in the BigTable including corresponding username, filename, file type, buffer start position, total file length, whether it is flushed or not, whether it is deleted or not and the corresponding flushed filename if flushed. And we support GET, PUT, CPUT, DELETE, RENAME, MOVE methods and we also allow flushing the memory content to files and also reload the flushed content to memory when reboot.

In our BigTable, we have memtable with a limit size in memory and when the user wants to put some files into our system, we first try to put it into our memtable and check whether it is exceed the limit size of the memtable, if so, we will flush all the memory content to disk just like the SSTables. Besides, when the user wants to delete a certain file from our system, we will delete it lazily first, which means we just mark it as deleted in our BigTable and then we delete it from our disk periodically by deleting all the files marked as deleted in our system.

When a certain node crash, our BigTable supports that flush all the data on that node in the current memory including memtable, metadata in bigtable, metadata for deleted files to the disk and also supports reload them from the disk to memory again when rebooting.

- SMTP Client

For the email service, we use res_query to retrieve the corresponding IP address of a desired email address domain and usually each IP address use port 25 for accepting SMTP email requests. And usually there are more than one IP addresses corresponding to a certain domain and we will choose the first one. Then after getting the IP address, we can send requests to that address and port. Finally, we can send emails to gmail.com successfully. Now it only works for gmail, connections to the addresses parsed from other service providers will time out.

Protocols

The protocols we used for communications are too many contents to include in this report, therefore details please see another document we send along with this report.

Discussion

Design Decision

- Encoding design:

We encode all file data into hex. The advantage is we can then handle complex file upload/download such as images, pdf, sound, videos. The disadvantage is that the data becomes larger after encoding hence more pressure is brought for socket communication.

- Front end design:

We used round robin algorithm for redirecting requests in load balancers, the advantage is that it is easier to implement and has less chance for conflict, when number of requests is not large, it can distribute requests evenly, however, if data is large, this algorithm wouldn't work well.

- Backend design:

We care consistency more than latency. So frontend will writes both primary and replica servers and only read primary servers. From perspective of quorum, W=2 and R=1. And we don't want our master to become an obvious bottleneck. Although master has to receive a lot of requests from other servers, all requests are short and simple. There will never be any long request(including file content, etc) sent to master.

Challenges

- Frontend design:

Parsing and encoding/decoding binary data: the challenges was to figure out a good way to support passing all types of files in our system without damaging the data, to do this we needed to figure out how to parse data in a generic way that it handles complex data file such as mp3 and mp4, and then for preventing unexpected data loss, we needed to come up with a way to encode the data so that we can safely store in the backend server while can easily get the data, we figured encoding to hex string being a good way, however the compromise was increasing data size.

- Backend design:

Fault tolerance and consistency are the most challenging part of the backend implementation. Since servers can crash at any time, it's hard to maintain consistency in multithreaded environment after recovery. So far we can recover correctly in most cases. But still 3 cases cannot be fully handled:

a) Crash happened when checkpoint or log is written will leave incorrect info in logs or checkpoints. Hence crashed servers cannot recover.

b) If the time between crash and recover is too long (larger than a checkpoint time), then the server after recovery will fall behind at least one checkpoint.
c) Since we can only handle data transmission around 15MB (actually can be tuned in codes) in a single transmission, if the log is too large, recovery will fail since the log cannot be sent from the working server to the restarted server.

Future improvement

- Better Consistency

Given more time, we can implement sequence number based consistency for storage servers. Now the requests from frontend are not assigned sequence numbers from the master. So there will be potential order inconsistency for storage servers in the same group when network delay is large.

- Persistent Connection

Now the connection between frontend and backend are not persistent. So each time when frontend sends request, it needs to contact master first for the destination backend address. The number of connections to master can be large in extreme cases hence master becomes a performance bottleneck for this system.

Labor Division

Jian Li: BigTable API, SMTP client.
Yang Yu: Master, Storage Server
Ziyu Chen: Admin Console, Upload/download, UI
Lijun Mao: Load Balancer, Web Server