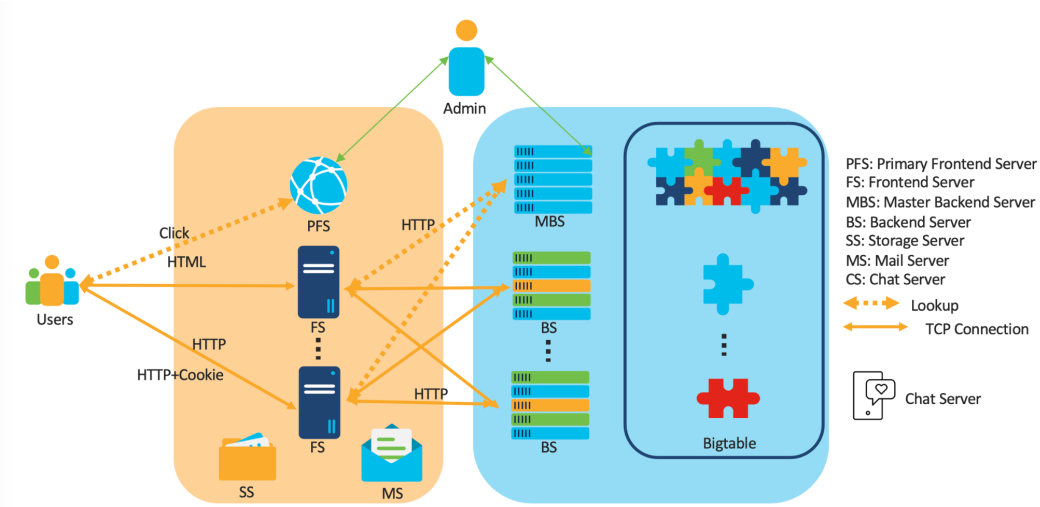
**Penn Cloud Report - Backend**

Group T24
Collaborator: Gaoxiang Luo, Jingyi, Li




## Responsibilities

**Gaoxiang Luo**: MasterBackend and MasterFrontend (load balancing, server liveliness), AdminConsole (status and raw data display)

**Jingyi Li**: Key-value Store, Load Balancing on the Backend, Data Transmission for Admin Console

## MasterBackend, MasterFrontend, Adminconsole - Gaoxiang Luo

**(masterbackendserver.cc/h, masterfrontendserver.cc/h, frontendserver.cc/h)**

**Load Balancing on MBS**: As the storage servers responsible for a range of row keys have their own replicas, a key challenge for the master backend server is to properly redirect frontend servers to a replica that is least-busy while all replicas are alive. So, we decided to evaluate the business of each replica via the number of connections it currently has, and we redirected frontend servers to a replica that is alive and has the least instantaneous connections. This is implemented by the master backend sending a unique message to all backends and receiving a number that stands for the current connection of that backend server.

**Drawback of MBS**: Single Point of Failure. Our assumption toward the submission of this project is that MBS won't crash. A natural solution is to make replicas for MBS; however, GFS slides 13-15 portray a very similar design of our bigtable, so MBS can also do log and checkpoint so it can recover from failure instead of replication.

**Server Status of BS in MBS**: As the admin console needs to know the current status of all backend servers, it would be too expensive and non-scalable if we have an admin console to check liveliness with each backend server. Hence, the master backend server would send ping each backend server periodically (e.g., 5 seconds) by trying to establish a TCP connection, and store the backend status (alive or down) in a local data structure that will be ready to be sent to the admin console when needed.

**Load Balancing on MFS**: Since FS is multi-threaded for concurrent users and TCP connections to their backend servers could be established within a thread, we use a vector of pairs that store the usernames as well as the socket fire descriptors of their connected backend servers. Therefore, a TCP connection with its backend server only needs to be established once before logout. Each FS stores a vector of active users. The load balancing of MFS can be implemented by checking each FS the current length of such a vector, and choosing the one with the least loading. This is also implemented by MFS sending messages to all frontend servers.

**Server Status of FS in MFS**: Originally, the MFS would check statuses with all FS during load balancing, so we planned to implement the server status checking together with load balancing and make it event-driven. However, then the server status on the admin console might be lagging than it really is. So, we also decided to periodically ping all FS, just like MBS.

**Binary Sending/Receiving Handling between FS and BS**: Originally, the data type that we used to store the value of bigtable cells was C++ string. However, the binary may contain null terminators which caused the data to be truncated unintentionally. Hence, we switched to the vector of char and sent the data by sequences of bytes, which solved the problem.

**Status and Keys Display**: Whenever the admin console likes to retrieve the usernames as well as their stored attributes, it would send a message to the master backend server with a TCP connection and receive all the row keys and column keys that are associated with each row key. Besides the keys display, the admin console would also reach out to master frontend and master backend server respectively to get the latest status of all frontend and backend servers.

## Key-value Store - Jingyi Li (storageserver.cc, storageserver.h)

**Overall Scheme of Data Storage**: All key-value pairs are stored in a nested map of std::vector<char>s. This will allow accessing the data in each cell by calling the row key and column key. Using a vector of char will also guarantee that binary data will not be truncated by the /0 termination rule of the c++ style std::string and c style c_str().

**Overall Schema of Data Transmission**: All data is transmitted using two primitives: writeWithSize(int, std::vector<char>*) and readWithSize(int, std::vector<char>*). These two functions implement the reading and writing algorithm in stream socket as introduced in lecture 6's slides under the hood. The write function will first calculate the size of the data stored in the std::vector<char> and then send the data size and the actual data to the other end of the pipe in sequence; the read function will first receive the size of the data it should expect and then keep reading until it has received all data. Such a scheme will make sure there will be no data loss during the transmission and the size of the transmitted data will not be limited to TCP's pipe (which is 65535 bytes).
(One subtle bug I found during the implementation of the above scheme was that the write() function will cause the program to crash under the situation of a broken pipe, substituting it with

send() function with a MSG_NONSIGNAL will help to catch the broken pipe error without crashing the program)

**Overall Architecture of Storage Server**: The storage server implements a multithreading architecture and stream socket. A new thread will be created to handle the newly-coming request. There is a limit of maximum thread defined, once a server receives a connection with index larger than the maximum thread, it will check if there is any thread whose connection has already been closed and that thread will be reused by the new connection. Another strategy to prevent thread counts from building up is that a global array of the connection status (0 for dead, 1 for alive) is kept and whenever a connection is about to be closed during data transmission between the backend and the frontend or among the replicas, a “QUIT” message will be sent to the storage server. The storage server will then clean up the resources of the current thread and also set the current thread’s connection status to 0 in the global variable so that the new thread will know that the current thread can be recycled.
Such design will also allow the master backend node to do load balancing where it needs to know the number of alive connections of each replica of the storage server in order to choose the “least busy” one for the client to establish a connection to.

**Overall Scheme of Replicas of Storage Server**: We chose a primary-based model to achieve consistency and fault tolerance among all replicas. There will only be one primary storage server at a time and a leader election process will be triggered once the current primary is down.

**Consistency among Replicas of Storage Servers**: all replicas will follow sequential consistency, which is achieved by implementing the remote write methods. If a request (of PUT/CPUT/DELETE) was received by the primary node, the request will first be propagated to all the secondary nodes. The primary node will only process the request and append it to logfile after receiving OKs from all secondary nodes and then send an OK message to the client, or else, if any ERR messages was sent to the primary node, the primary node will not execute the request on itself and and ERR message will be returned to the client. If a request(of PUT/CPUT/DELETE) was received by a secondary node, the request will first be sent to the primary node and the above process will be repeated. Once the primary node makes sure that all replicas have successfully executed the request, it will send back an OK message to the current secondary node and an OK message will be sent to the client. If any error occurs in any of the replicas, the client will receive an ERR message.
For other types of requests like GET and other inter-server communication messages will not go through all replicas, instead, they will only be handled on the local node.

**Leader Election:** Each replica will allocate a thread to send heartbeat messages to the current primary node to check its status, once the failure of primary node is detected, a leader election will be triggered among the rest of the replicas. I chose to implement the bully algorithm for leader election: once a node cannot connect to the primary node, it will send a “LEADERELECTION” message to all nodes with higher index. If a node does not receive any message in the process, it indicates that it has the smallest index and it should be elected as the new primary node. Then such node will then send “DECIDEPRIMARY <primary_index>” to

all the other nodes. Once upon receiving such a message, a node should update the primary index and start to send heartbeat messages to the new primary node.
Using the bully algorithm is scalable for systems of this type because it will always choose the correct node with the smallest index to be the primary node.

**Checkpointing and Logging**: The checkpointing file and the logging file are kept for a tablet stored on a node. Everytime after a successful execution of the following request: PUT, CPUT and DELETE, it will be appended to the logfile and the sequence number of the request will be incremented by one. A thread is allocated to do periodic checkpointing. Each checkpointing will increase the version number of the checkpoint by one. Everytime after checkpointing, the logfile will be cleared and starting to record new requests coming after the most recent checkpoint.

**Recovery from Failure**: When any of the replicas is brought back after failure, it will first contact all the other replicas for three types of data: the checkpoint file, the logfile and the primary index. In the meantime, a “RECOVERYSTART” message will be sent to other replicas to suspend them from execution. The current node will first update the primary index based on what’s received and pick the checkpoint file with the highest version number to load into its current data, and then pick the logfile with the highest sequence number to replay all requests recorded in it. It will then send out “RECOVERYEND” message to other replicas so that all nodes can keep running once the current node finishes its recovery. This will make sure that the current node can keep everything updated after a failure.

**Locking and Synchronization**: Locks are associated with tablets and only with critical session when the data stored on the node actually get edited to allow maximum concurrency. Lock is also used to suspend the current program from execution when another replica is recovering from failure.

**Support For Admin Console**: The storage server will send to the master backend server all the key-value pairs upon receiving a “METADATA” message. This data will then be displayed by the Admin Console’s UI to visualize data stored in each replicas’ group.

**An Overall Challenge in Storage Server Implementation**: Storage server is about manipulating data on BYTE level to a large extent. So multiple tricks were devised to make sure of the correctness and efficiency. Apart from choosing std::vector<char> as data type and using the writeWithSize and readWithSize primitive for data transmission, the byte count of the data of each cell is also included in the checkpoint file and the logfile so that when recovering from a file and replaying requests, the program can always read data with the correct count of bytes.