Design Decisions and Features

Key-value store

The key value store uses a distributed tablet based design, where data is partitioned across tablets and servers via hashing of the row and column, with nodes supporting puts, gets, cputs, and deletes. Internally, reads access the Memtable first, an in memory sorted map holding recent writes, that is flushed to disk once it reaches a certain capacity. If the requested data is not found in the Memtable, the request then goes to the SSTables, which are immutable sorted on disk files created from flushes of the memtable. There are potentially many SSTables, and so a bloom filter is used to query the SSTable for a "definitely not in set" or "maybe in set" test. Writes go first to the WAL (write ahead log), where they are saved to disk prior to being applied, to ensure that mutations are able to be recovered in the case of a crash.

Replication

This system uses primary secondary replication, with the coordinator serving as the definitive source of truth for which node is primary and handling new primary changes in the case of nodes going down. Writes are only allowed to go to the primary, and when they do, they are logged locally, applied, and then asynchronously forwarded to all secondary replicas.

Failure Detection

The system uses a heartbeat protocol, where the coordinator sends periodic heartbeat pings to workers via a background thread, and marks nodes as suspected dead, and then dead after a configurable period of time has passed. Once a node is marked as dead, the coordinator will trigger its recovery procedures for the tablets hosted by the node. The coordinator also updates the tablet mapping, so that if clients request a tablet mapping, they won't be redirected to a down node.

Recovery

When a node that was previously down is restarted, it will attempt to register with the coordinator again, and for each tablet stored locally, the coordinator will mark it as a secondary, unless there are no live replicas in which case the node will be marked as primary. Upon this tablet assignment, the node will check if it has the tablet data on disk, and will then load the existing tablet and replay the WAL, to restore the memtable. If other replicas are live, the node will reach out to the primary and attempt to sync. The recovering node will send its latest version number and then the primary will respond with all entries with versions greater than that, and the recovering node will apply those entries to catch up to the current state.

HTTP-server and Load-balancer:

Initial requests to the domain on port 8080 are routed to a load balancing server. Using a round-robin scheme, clients are assigned an active and enabled http server. HTTP servers are active if they are responding to heartbeat requests and enabled if they are not disabled by administrators through a basic command interface. Clients are then redirected to the chosen

HTTP server and requests are from then on handled by it. The HTTP server uses an internal load balancer to distribute requests to router processes on the node. Each router process parses the clients request, extracting the method, path, headers, and body of the request. Based on the path the request is sent to a specific service or handled as a static file request. Additionally, only certain pages are accessible based on the authentication status of the client, so cookie headers are checked to determine more complicated rerouting.

Login/Auth
Authentication status is determined using cookies. Usernames and passwords are stored within the KV backend under a special key for the user. Non-authenticated users only have access to a home/login page to create or log in to an account, and are redirected to this page from all others. Once authentication is determined with a cookie, users have access to their emails and storage and the account page to change passwords

File System
The file system is built on top of the key-value store. Each row corresponds to a user. Each user has a ".files" column, and the data in this column is a list of files, where each line contains the filename, the directory the file is housed in, and whether the file is a directory or a file. Each non-directory file also contains an entry in the KVS under the column "{path}\{filename}". These keys are guaranteed to be unique from each other and from the ".files" column because no filename can contain a period, and each filename must be distinct from any other file within the same directory. Username also must be distinct, so the row keys are distinct. The UI pulls from the entry in the ".files" column to show all files in the current directory; therefore, all file operations are done by sending an HTTP request to manipulate the entry in the ".files" column to reflect the updated file system (the exception is downloading which does not change the state of the file system and only sends the file data back through an HTTP response). Uploading and moving/renaming non-directory files perform operations on the KVS outside of the ".files" column by creating a new entry under the column "{path}\{filename}" (and deleting the old entry in the case of moving/renaming). The file system supports files of up to 10.4 MB, which was arbitrarily chosen because of the >10 MB requirement and could easily be increased if necessary.

Email Server

The mail server consists of three main components: an SMTP server, a relay client, and a webmail interface, all sharing the key-value store backend. The emails will be stored with all of their data - ID, parent ID (0 if no parents), from/to addresses, subject, and body.

We implemented both an 'inbox' and a 'sent' mail section, which are stored as a list of unique IDs for each user's emails.

The SMTP server receives incoming messages, delivering local mail directly to the recipient's mailbox entry in the backend. The SMTP server is run by the main HTTP server, and if a replica sees that the port (2500) is blocked, it starts a SMTP heartbeat thread. If the port becomes unblocked for whatever reason, one of the other replicas will detect it and start the SMTP server.

For sending external emails, we first see if a mail address has an ‘@’, and then our SMTP client queries DNS to find the MX records for the recipient's domain and connects to the server to send the mail.

Replying to emails will open the compose modal with the TO address and subject prefilled and not changeable. Forwarding emails will open the modal, with the email content prefilled and the subject prepended with ‘FWD: ‘.

Admin Console

The admin console allows you to view the backend data (paginated), shut down and restart frontend and backend nodes.

Using a simple command interface on the load balancer node, http/frontend servers can be requested and shown in the console. Additionally, administrators can disable/enable servers. This does not kill the underlying server node, but simply prevents the load balancer from assigning new clients to the node.

Extra Credit

Aws Deployment

The system was designed from the start to not hardcode localhost and to specify hostnames when necessary via the command line, so for deploying to AWS, several shell scripts were used, first to launch all required EC2 instances (t2.micros were used), then to setup the environment (to make the code locally), and third to launch the actual processes. The security groups were configured to allow intercommunication between instances on the internal ports used for heartbeats, KVS data, etc. Whenever possible, code was written to avoid needing to be invoked in a particular order, so that deployments could be made easier.

Raft log

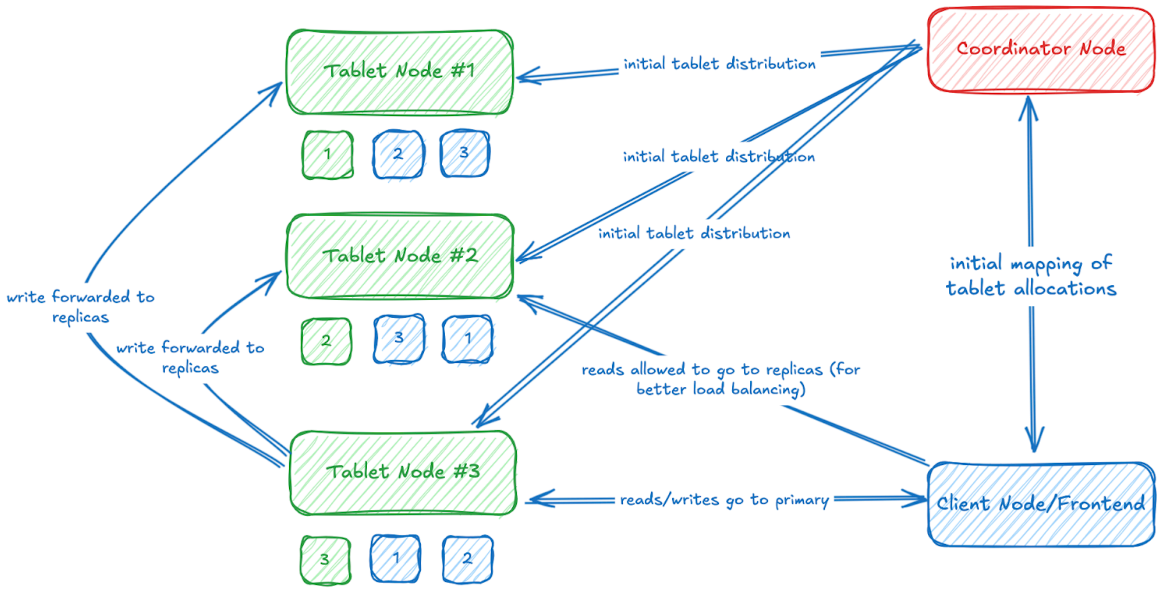
Raft is used to manage the state of a cluster of coordinators. The raft log contains a numbered list of entries that can be used to deterministically restore the state of a node. It is replicated across a majority of nodes before any state change can be applied to the leaders, so that ma ajority consensus can be reached. There are 2 main RPC functions that are used to implement Raft: AppendEntries and RequestVote.

When a state-altering command is issued to the leader coordinator, it writes a corresponding log entry in its local log and sends AppendEntries with the new log entry to each follower. For each follower, the leader waits for an acknowledgement. When over half of the followers have acked, the leader will commit the entry, which causes it to be executed. Otherwise, the leader node waits. In the case that the majority of followers don't acknowledge the RPC before another leader is elected, the new leader will override the unacknowledged entry since it hasn't been committed. This guarantees that the majority of nodes have the same data in their logs. When a follower receives an AppendEntry, it'll check if the term number is up to date, and if the last log entry's term and entry number match. If not, it'll send a failure message and ask the leader to send an extra log entry until the starting term numbers and entry number of the logs match.

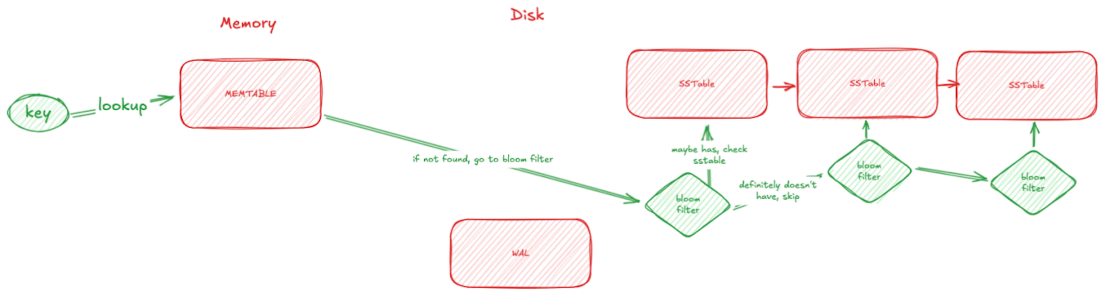
The RequestVote RPC is used to elect the leader of the cluster. Each node has a heartbeat sending thread, which sends periodic heartbeats to all other nodes in the cluster if it's a leader. Each node also has a heartbeat checking thread, which checks when the last heartbeat was received. If the last heartbeat was received before a certain timeout limit, the current node will update its status from Follower to Candidate, increment its term number, and send RequestVote to each other coordinator. Each other coordinator will accept the vote if the candidate's term number is greater than or equal to the current term number. If the Candidate node has a majority of followers accept its RequestVote RPC, it'll become a leader and send an empty AppendEntries to notify other nodes of the change. There's also a separate file containing the term number, commit index, and voted_for value for each coordinator. This helps recover the state in case of a crash.

The raft log is fully functional, with a majority of coordinator logs matching at any given time, and all of them matching eventually, but deterministic replay doesn't work. This is because we logged function calls from the handle_request function in the coordinator event loop, but there are 2 problems: Some of the functions are asynchronous and depend on a response from the kvs, and some state-modifying functions are called outside of the main event loop via helper threads. If we had more time, we'd isolate all of the state-modifying parts of the code into independent functions and log each of those. The raft log is also not merged into the main (present in branch - coordinator-cluster-consensus) since it modifies the functionality of the coordinator.

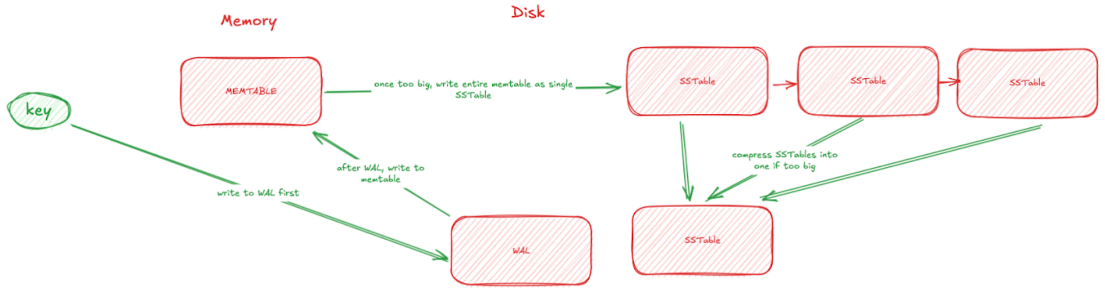
Overview of Architecture


Tablet Node #1
1
2
3
Tablet Node #2
2
3
1
Tablet Node #3
3
1
2
Coordinator Node
Client Node/Frontend
initial tablet distribution
initial tablet distribution
initial tablet distribution
initial mapping of tablet allocations
reads allowed to go to replicas (for better load balancing)
reads/writes go to primary
write forwarded to replicas
write forwarded to replicas

Internal Read



Internal Write


Memory
Disk
key
MEMTABLE
once too big, write entire memtable as single SSTable
SSTable
SSTable
SSTable
write to WAL first
after WAL, write to memtable
WAL
compress SSTables into one if too big
SSTable