RaftKV: A distributed key value store with fault-tolerance support

Amy Liu, Jerry Liu, Kehong Liu, Zipeng Liang, David Miller, Andrew Sung

The changes we added to address the comments are highlighted

Introduction and Background

With the rapid development of distributed systems, one of its most widely used applications is distributed components in data storage. A distributed key value store is a common structure of such a system. One of the design challenges is to handle the scenario where part of the system might not be working reliably.

In the project, we present a distributed key value store which uses Raft as backbone support for providing consensus and failure handling. The application will handle common failures like network partition or node failure, and only fail to respond when the majority of all working servers fail. It should also be able to handle new server join or a scenario of rejoining when previously failed servers are back online.

System Architecture

We chose to have an all-to-all connection between servers. This is for the convenience of the Raft process where servers need to communicate with each other.

Each server and each client is identified by a unique serverID/clientID, none of the 2 clients/servers should share the same id.

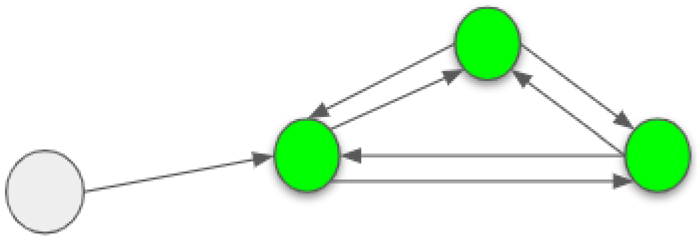

Fig 1. Green nodes are the servers and gray node is a client

We wait for all servers to join before clients send ops (like in A3). After the initial joining phase, servers can fail and restart.

Clients do not need to maintain a localOpId in this system since all operations will be globally ordered by the coord. Clients can also send GET/PUT requests to any node, and these requests will then be forwarded to the coordinator node for handling.

Assumptions we can make:

Change: Removed the new server join part because of its difficulty on implementation

1. There will be N known servers
2. Servers may fail and lose connection
3. Servers will not fail before coord has logged the event SystemReady
4. Data on disk will not be corrupted or lost
5. The logs will *eventually* be consistent
6. The system will always have at least N/2 + 1 servers up and running in the same partition so that the entire system will never fail

Assumptions we cannot make:

1. The logs are *always* consistent across all servers
2. All requests issued by the clients will be handled
3. The system is always available

Leader as Coordinator

For the simplicity of ordering all the clients' requests and making sure the state is consistent, there will be a coordinator server to handle all requests. The client is allowed to connect to any arbitrary server when first connecting to the system, but all requests must be sent to the coordinator server.

The coordinator server will have the storage state (most likely be a simple map of key and value), and all the GET requests will be looked up in the coordinator's storage state. Once a response is confirmed via consensus, the server connected to the client can send the response back to clients.

At the initial state of the system, there's no coordinator yet. All the servers are followers and they will generate a time counter with random timeout locally. If any server has a timeout, it will become a candidate. It will increase the current term number and raise a new round of election by broadcasting the request for voting to all the other servers.

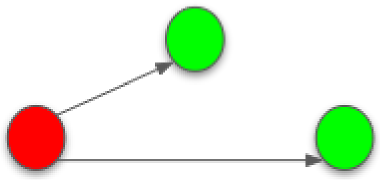

Fig 2. Timeout and broadcast

As stated by fig 2, red node timeout and broadcast a request vote with a specific term number. As a follower (green node), when they receive such a voting request before timeout, the request is then replied to by a vote. If the candidate node (red node, or the node proposes the election by nominating itself) receives more than the majority of votes (N/2), it is elected as a new coordinator. The vote request is labeled by the term number, which should monotonically increase every time a new round of election is proposed. If multiple vote requests with the same term number are received, only the earliest one will be replied with a vote.

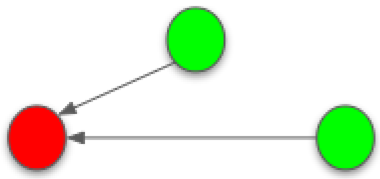

Fig 3. Receive vote from peers

Once a coordinator is elected, it then broadcasts to the rest of the servers about the situation. All the other servers will be in follower state and reset their timer to listen to the heartbeat message from the coordinator. This can be done by using fcheck from previous assignments or other heartbeat library which serves a similar purpose.

If no coordinator is elected in one term, it's typically the case where the election timeout before receiving the needed votes or being notified by the elected leader. In this case, the server with timeout should increase the term number and propose itself as the coordinator again to start a new round of election.

At this point, all the servers will be able to handle requests by simply forwarding the requests to the coordinator. Before a coordinator is elected, all requests sent from clients will be queued until the election is over.

Local State Machine and Replica

Server may crash in an unpredicted way, including the coordinator server. A common way to handle this kind of failure is by sharing state among multiple different working nodes, so that it won't be completely lost if only part of the system dies. This is done in raft by doing log replication, which will be adopted in our system as well.

The **Log** in our system is similar to a regular raft log. Each log entry contains a log index, the term number, and the operation/request that we share across the servers. The log will be persisted into disk for each server for further use in failure recovery.

The log is only appended when the coordinator issues an AppendEntry request to all of its followers. The coordinator will not be able to overwrite the existing entry of the log but can only append the log unless in the special case of failure recovery, which will be discussed later. Therefore, after receiving a request, the coordinator will do the following: accept the request -> call AppendEntry to append in local log -> issue AppendEntry event to all followers. Once the follower receives the AppendEntry event, they also append the entry to their local log and then send back a confirmation response to the coordinator. Just like the election, if the coordinator receives confirmation from the majority of its followers, the operation specified by the entry will be applied to its local state machine. In our case, the state machine is likely to be a storage state with key and value implemented by map. The request is then considered committed.

The commit event is then sent to all the followers and they will apply the same entry to their local state (similar to what we have in the coordinator) so that logs are consistent throughout the entire cluster.

Possible Problems

There are many potential problems that may come out when the servers start to fail unpredictably and also because of the asynchronous network. We will show case-by-case that most of them will be solved by the safety insurance provided by raft.

P1: Follower failure

One failure case we want to handle is when one or more followers are unreachable, regardless of why they are (network partition or the server just fail). In this case, the coordinator will simply retry the AppendEntry request indefinitely, until the follower is back online and reply with confirmation, or if the coordinator simply is replaced by another coordinator election. The rest of the system will not be affected.

Since the logs are persisted into the disks, they will survive the server crashes. When a previously failed server is back online, it should first read the log and apply all the committed entries to restore the state to the latest consensus. The server should also send a request to the coordinator for any log entries that the server may have missed out on while it was offline.

P2: Coordinator failure

The coordinator may fail just like any other server; this will result in a new round of election. However, when coord failure occurs, there’s no guarantee that the log for the previous coordinator is successfully replicated in the majority of the followers.

The coord sends periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no heartbeat from the coord over a period of time called the *election timeout*, then the follower assumes there is no viable leader and begins an election to choose a new leader.

When a new coordinator is elected, it will first need to repair the potential log inconsistency. This is the case we mentioned before where the coordinator can overwrite the logs of the followers if the log in the coordinator is not consistent with the log in the follower. To do so, for each of its followers, the coordinator will broadcast its log to all of the followers. The followers will then find the last entry where they agree, then delete all the entries coming after this critical entry in the follower log and replace them with the coord’s log entries. This includes appending any entries from the coord’s log that the server never received. This mechanism will restore log consistency in a cluster subject to failures.

It’s also possible that the newly elected coordinator has less log entries replicated and therefore the repair process may cause some followers to lose its state even if they are more up to date. To avoid that, we always try to elect the coordinator with the most complete log. Therefore we introduce an *election restriction check*. During elections, choose candidate with log most likely to contain all committed entries Candidates include log info in RequestVote RPCs (index & term of last log entry) voting server V denies vote if its log is “more complete”: (lastTermV > lastTermC) || (lastTermV == lastTermC) && (lastIndexV > lastIndexC). Then the coordinator will have the “most complete” log among the electing majority.

Todo: What happens when the majority of servers fail (together with coord)? → remaining servers elect a new coord, but then coord should realize that the remaining servers do not constitute a majority. (i.e. If 50 servers were in the system and 26 die, 24 remain. The newly elected coord must detect that 24 < 50/2 and thus should broadcast that the system should terminate.

P3: Multiple Elections

To avoid concurrent elections, Raft ventures for a simple solution — randomized timeout. The key idea here is that randomization reduces the probability of election collision. If a split vote happens, we can simply retry with a very large probability of getting it done the next round.

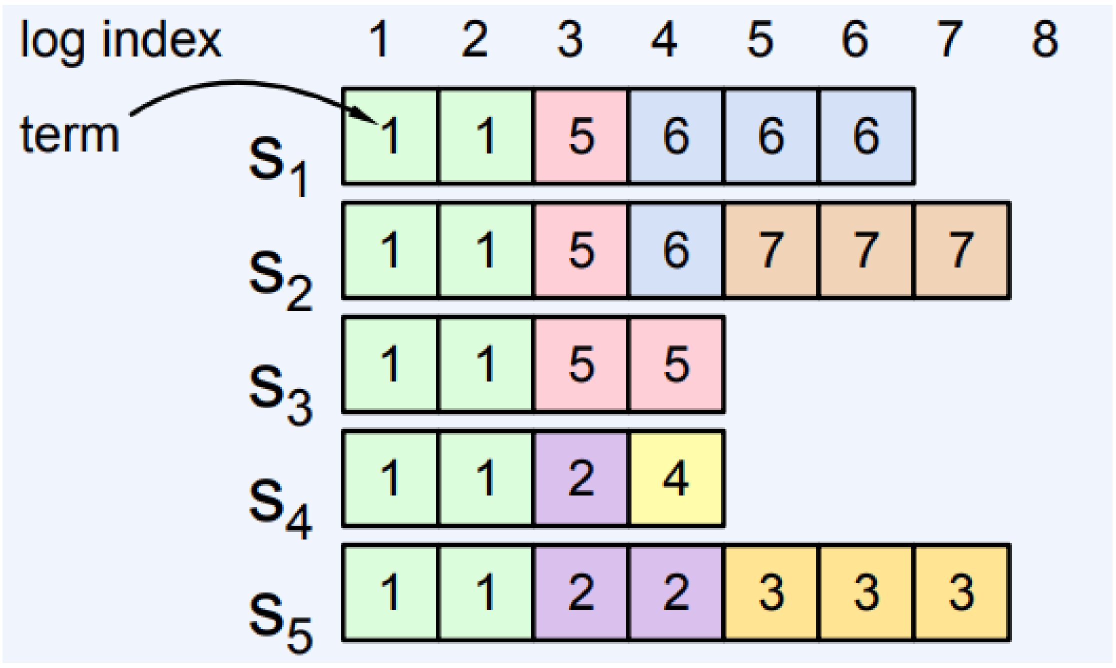

Fig 4

Suppose we have 5 servers and it’s shown as fig 4, S4 and S5 won’t be able to be elected because their logs have less term number than the majority of the servers.

P4: Multiple Leaders and leader rejoin

Leader in Raft is equivalent to the coordinator in our system. If an old coordinator is cut off, it won’t be able to reach the majority. Therefore, even if it identifies itself as a coordinator, it’s actually out of the system. The rest of the system will elect a new coordinator to replace it.

When the old coordinator rejoins the network successfully, it will again get the access to reach the majority. However, when a new election is conducted, the candidate increases the Term number by 1. The updated term will be propagated to all nodes who voted for the new coordinator, which are guaranteed to be the majority. Now, if the old coordinator rejoins the group, it’ll discover that it has a lower Term number — time to step down. The same principle applied even if the coordinator is always in the system: if a new coordinator is elected and broadcasted with a larger term number, the old coordinator with less term number will be reverted back to a normal follower.

API Design

The following section list the rough API design of our system, they are categorized into Client, Server, Raft:

1. Client:

a. *MakeClient(servers []*rpcutil.ClientEnd) *Client*
b. *(c *Client) Start(key string) notify-channel, err*
c. *(c *Client) Get(key string) err*
d. *(c *Client) Put(key string, value string) err*
~~e. *(c *Client) ServerRejoin(args *ServerJoinArgs, reply *ServerJoinReply) error*~~
~~f. *(c *Client) ServerFail(args *ServerFailArgs, reply *ServerFailReply) error*~~

The API for the client is mostly the same as a3 except with some extra rpc endpoints for the server to notify the client about different situations.
~~*ServerFail* is when the current server that the client is connected to loses its connection with other servers but it's still connected to the client. In this case the server should notify the client about it so that the client will try to connect to a different server.~~

The client will not be notified about the details of raft election or new coordinator; the use of a coordinator is transparent to the client. Instead, the client's requests will be forwarded by the server it is directly connected to (this can be any server in the cluster including the coordinator itself) to the current coordinator. If a new coordinator is elected, then the requests will be forwarded to the new address by the server. All the details are hidden from the client.

2. Server:
   a. *Start(serverId uint8, serverAddr string, serverListenAddr string, serverHeartbeatAddr string) error*
   b. *(kvs *KVServer) Get(args *GetArgs, reply *GetReply) error*
   c. *(kvs *KVServer) Put(args *PutArgs, reply *PutReply) error*
   d. *(kvs *KVServer) NewServerJoin(args *ServerJoinArgs, reply *ServerJoinReply) error: RPC endpoint for server joins. Called by the newly joining server when it notifies this server that it is joining the system.*

3. *Raft:*
   a. *(rf *Raft) GetState() RaftState:* Get the current state of raft, including the leader/coordinator id, term, latest commit index and possibly other info
   b. *(rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) error:* Request vote rpc endpoint, election restrict check should be done here
   c. *(rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) error:* AppendEntries rpc endpoint, called when the coordinator issue the AppendEntries event
   d. *Start(peers []*rpcutil.ClientEnd, me int, persister *Persister, applyCh chan ApplyMsg) *Raft:* Starting endpoint of the entire raft package

e. *(rf *Raft) NewServerJoin(server *rpcutil.ClientEnd) error*

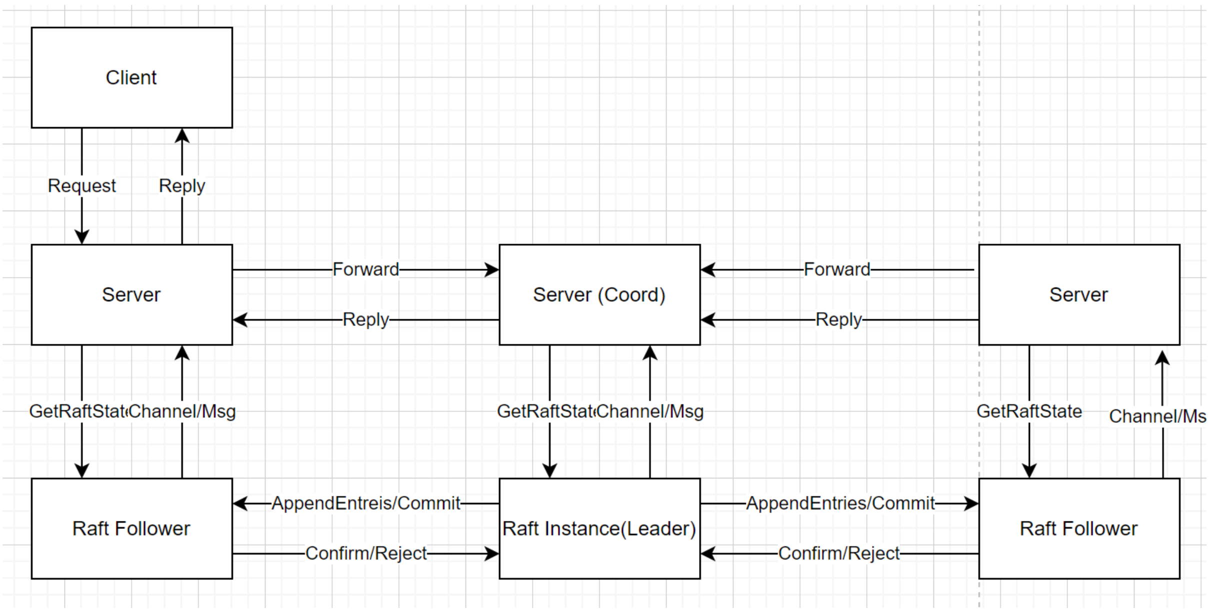

Fig 5 Arch Diagram

Fig 5 shows the rough architecture of our system. The arrows represent the operations one instance can perform by calling the API of another instance. For example, we can imagine each entity of the figure as a struct, and that they communicate with each other by calling the API defined in their structs. The arrows between raft instances are the raft functionality, such as AppendEntries and Commit, which are used to ensure the consistency of the system.
The diagram also shows how, as stated earlier, client requests are forwarded directly from the server contacted by the client to the coord server. The coord communicates to the other servers (raft followers) via its raft instance to ensure consistency, then it returns the response to the server from which it received the request. Finally, the server which initially received the request from the client returns the response to the client.

Timeline

March 18
- Project proposal drafts

March 25
- Final project proposals - after approval, distribute tasks

April 1 (M1)
- Implement the raft mechanism (Jerry, Kehong, Zipeng)
- Implement the stable server node (Andrew, David)
- Implement the client node (Amy)

April 4-8
- Project update meetings

April 8 (M2)

- Integration of raft, server, and client - happy path working

April 15 (M3)
- Implement external server join process
- Test server failure and recovery

April 19
- Project code and final reports

April 18-22
- Project demos

Milestones

- M1: Implement the raft functions which includes propose, voting, appendEntries, commit. Implement a kv server cluster which can communicate with each other and forward requests to the coordinator. Implement client nodes.
- M2: Integration of raft and server-client. Happy path should be working.
- M3: Implement failure recovery (reapply the operations persist in log), new coordinator log repair. External server join (server that's not in the original set).

Evaluation Methodology

- M1: System should work when no failures occur. Gets/Puts should be serviced in some serializable order~~, the system should be able to elect 1st coord and run until N/2 + 1 servers are stopped~~. Failure recovery is not required at this stage. For this milestone, we can assume the coord and other servers will not fail.
    - Traces: Server join (for system initialization), Get, Put

- M2: The raft should be integrated into the servers. The servers will be able to forward the requests to the coordinator and start a new round of raft election when the coordinator dies.
    - Traces: Coord election (initial coord), Raft log traces

- M3: The failure recovery should be implemented. When a server is back online, it should be restored to the state before crashes and further synchronized with the coordinator.
    - Traces: ServerFail, Server join (re-join), CoordFail, ServerFail, Election tracing (for new coords)

Tracing

We will use tracing to help ensure correctness and visualize system executions. Many of these trace actions are similar to those of a3.

- Server tracing (strace)
    - ServerStart{*serverId*}: Recorded in strace. Signifies the start of a server node
    - ServerJoining{*serverId*}: Recorded in strace of the joining server. Indicates that the server with id *serverId* is about to join (or re-join) the system.

    - ServerJoined*{serverId}*: Recorded in strace of the joining server. Indicates that the server with id *serverId* has joined (or re-joined) the system.
    - CoordFail*{serverId}:* Recorded in strace. Indicates that an election timeout has occurred in the server. This will prompt a new election.
- Coord tracing (ctrace)
    - ServerFail{*serverId*}: Recorded in ctrace. Indicates that the coord detected failure of server with id *serverId*.
    - SystemReady*{}*: Recorded in ctrace. Indicates that (1) N servers have joined the system, (2) the first coord server has been elected, and (3) the KVS is ready to process client operations.
- Coord Election tracing (etrace)
    - Proposal*{serverId, term}:* Recorded in etrace. Indicates that an election timeout has occurred, prompting a server to start a new election with itself as a leader. Note that this will be followed by a single VoteConfirm{serverId, serverId, term} since the will vote for itself.
    - VoteRequest*{serverId, term}:* Recorded in etrace. Indicates that a coord candidate has requested a vote for its proposal.
    - VoteConfirm*{serverId, candidateServerId, term}:* Recorded in etrace. Indicates that a server has voted for the proposal {candidateServerId, term}. Must be preceded by a VoteRequest{candidateServerId, term}
    - VoteReject*{serverId, candidateServerId, term}:* Recorded in etrace. Indicates that a server has against the proposal of {candidateServerId, term}. Must be preceded by a VoteRequest{candidateServerId, term}
    - ProposalAccepted*{serverId, term}:* Recorded in etrace. Indicates that a proposal has received a majority (N/2+1) confirmation votes. Must be followed by CoordElected{serverId}
    - ProposalRejected*{serverId, term}:* Recorded in etrace. Indicates that a proposal has not received a majority of confirmation votes.
    - CoordElected*{serverId}*: Recorded in etrace. Indicates that a coord candidate has received a majority of confirmed vote for its proposal, and it has become the new leader.
- Raft log tracing (ltrace)
    - AppendEntry*{term, operation}:* Recorded in ltrace. Indicates that the coord is proposing a new entry to the log
    - AppendConfirm*{term, operation}:* Recorded in ltrace. Indicates that a server approves adding {term, operation} to the log.
    - AppendReject*{term, operation}:* Recorded in ltrace. Indicates that a server rejects adding {term, operation} to the log.
    - CoordCommit*{term, operation}*: Recorded in ltrace. Indicates that a coord has committed {term, operation} to its log. A CoordCommit must be preceded by at least N/2+1 AppendConfirm{term, operation} actions.
    - ServerCommit*{term, operation}:* Recorded in ltrace. Indicates that a server has committed {term, operation} to its log. Every ServerCommit should follow a CoordCommit{term, operation}.
- Client tracing (ktrace)
    - KvslibStart{*clientId*}: Recorded in ktrace. Signifies the start of a client's kvslib. For each unique *ClientId* corresponding to a running client, the *tracing log as*

a whole should contain this action exactly once, as part of ktrace. This action should happen-before all other actions recorded by the client with *ClientId*.
    - KvslibStop{*clientId*}: Recorded in ktrace. For a specific *clientId*, kvslib records this action to mark the end of a client session, corresponding to the completion of KVS.Stop(). For each *ClientId* corresponding to a running client, *the tracing log as a whole* should contain this action exactly once, as part of ktrace. This action should happen-after all other actions reported by a given *clientId*.
- Client operation tracing (get and put traces)
    - PutRecvd{*clientId, key, value*}: Recorded in put trace. Recorded by a server that is connected to the client (at the time of recording this action) and indicates that it has received a put operation from the client.
    - PutFwd{*clientId, key, value*}: Recorded in put trace. Recorded by a server just before it forwards the corresponding put operation to the coord server.
    - PutFwdRecvd{*clientId, key, value*}: Recorded in put trace. Recorded by a coord when it has received the corresponding put operation from a server.
    - PutResult{*clientId, key, value*}: Recorded in put trace. Recorded by the server that replies with a confirmation to the client about the corresponding put operation.
    - GetRecvd{*clientId, key*}: Recorded in get trace. Indicates that the (tail) server has received the corresponding get operation from the client.
    - GetResult{*clientId, key, value*}: Recorded in get trace. Recorded by the (tail) server that replies to the client with result of executing the corresponding get operation

References

1. Anon. The raft consensus algorithm. Retrieved March 25, 2022 from https://raft.github.io/
2. Anon. Designing for understandability: The raft consensus algorithm. Retrieved March 25, 2022 from https://raft.github.io/slides/uiuc2016.pdf
3. Diego Ongaro and John Ousterhout (2014). In Search of an Understandable Consensus Algorithm (Extended Version), June 2014.

RaftKV: A distributed key value store with fault-tolerance support

Amy Liu, Jerry Liu, Kehong Liu, Zipeng Liang, David Miller, Andrew Sung

The changes we added to address the comments are highlighted

Introduction and Background

With the rapid development of distributed systems, one of its most widely used applications is distributed components in data storage. A distributed key value store is a common structure of such a system. One of the design challenges is to handle the scenario where part of the system might not be working reliably.

In the project, we present a distributed key value store which uses Raft as backbone support for providing consensus and failure handling. The application will handle common failures like network partition or node failure, and only fail to respond when the majority of all working servers fail. It should also be able to handle new server join or a scenario of rejoining when previously failed servers are back online.

System Architecture

We chose to have an all-to-all connection between servers. This is for the convenience of the Raft process where servers need to communicate with each other.

Each server and each client is identified by a unique serverID/clientID, none of the 2 clients/servers should share the same id.

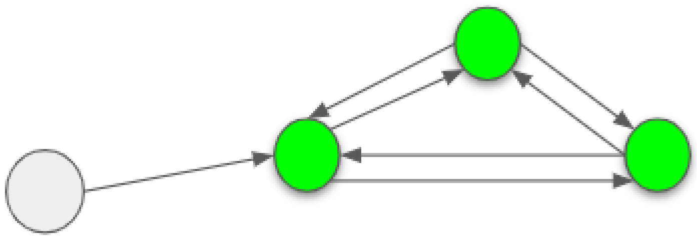

Fig 1. Green nodes are the servers and gray node is a client

We wait for all servers to join before clients send ops (like in A3). After the initial joining phase, servers can fail and restart.

Clients do not need to maintain a localOpId in this system since all operations will be globally ordered by the coord. Clients can also send GET/PUT requests to any node, and these requests will then be forwarded to the coordinator node for handling.

Assumptions we can make:

Change: Removed the new server join part because of its difficulty on implementation

1. There will be N known servers
2. Servers may fail and lose connection

3. Servers will not fail before coord has logged the event SystemReady
4. Data on disk will not be corrupted or lost
5. The logs will *eventually* be consistent
6. The system will always have at least N/2 + 1 servers up and running in the same partition so that the entire system will never fail

Assumptions we cannot make:

1. The logs are *always* consistent across all servers
2. All requests issued by the clients will be handled
3. The system is always available

Leader as Coordinator

For the simplicity of ordering all the clients' requests and making sure the state is consistent, there will be a coordinator server to handle all requests. The client is allowed to connect to any arbitrary server when first connecting to the system, but all requests must be sent to the coordinator server.

The coordinator server will have the storage state (most likely be a simple map of key and value), and all the GET requests will be looked up in the coordinator's storage state. Once a response is confirmed via consensus, the server connected to the client can send the response back to clients.

At the initial state of the system, there's no coordinator yet. All the servers are followers and they will generate a time counter with random timeout locally. If any server has a timeout, it will become a candidate. It will increase the current term number and raise a new round of election by broadcasting the request for voting to all the other servers.

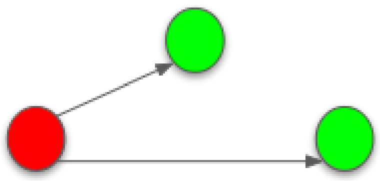

Fig 2. Timeout and broadcast

As stated by fig 2, red node timeout and broadcast a request vote with a specific term number. As a follower (green node), when they receive such a voting request before timeout, the request is then replied to by a vote. If the candidate node (red node, or the node proposes the election by nominating itself) receives more than the majority of votes (N/2), it is elected as a new coordinator. The vote request is labeled by the term number, which should monotonically increase every time a new round of election is proposed. If multiple vote requests with the same term number are received, only the earliest one will be replied with a vote.

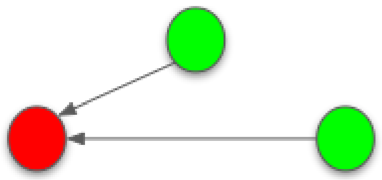

Fig 3. Receive vote from peers

Once a coordinator is elected, it then broadcasts to the rest of the servers about the situation. All the other servers will be in follower state and reset their timer to listen to the heartbeat message from the coordinator. This can be done by using fcheck from previous assignments or other heartbeat library which serves a similar purpose.

If no coordinator is elected in one term, it's typically the case where the election timeout before receiving the needed votes or being notified by the elected leader. In this case, the server with timeout should increase the term number and propose itself as the coordinator again to start a new round of election.

At this point, all the servers will be able to handle requests by simply forwarding the requests to the coordinator. Before a coordinator is elected, all requests sent from clients will be queued until the election is over.

Local State Machine and Replica

Server may crash in an unpredicted way, including the coordinator server. A common way to handle this kind of failure is by sharing state among multiple different working nodes, so that it won't be completely lost if only part of the system dies. This is done in raft by doing log replication, which will be adopted in our system as well.

The **Log** in our system is similar to a regular raft log. Each log entry contains a log index, the term number, and the operation/request that we share across the servers. The log will be persisted into disk for each server for further use in failure recovery.

The log is only appended when the coordinator issues an AppendEntry request to all of its followers. The coordinator will not be able to overwrite the existing entry of the log but can only append the log unless in the special case of failure recovery, which will be discussed later. Therefore, after receiving a request, the coordinator will do the following: accept the request -> call AppendEntry to append in local log -> issue AppendEntry event to all followers. Once the follower receives the AppendEntry event, they also append the entry to their local log and then send back a confirmation response to the coordinator. Just like the election, if the coordinator receives confirmation from the majority of its followers, the operation specified by the entry will be applied to its local state machine. In our case, the state machine is likely to be a storage state with key and value implemented by map. The request is then considered committed.

The commit event is then sent to all the followers and they will apply the same entry to their local state (similar to what we have in the coordinator) so that logs are consistent throughout the entire cluster.

Possible Problems

There are many potential problems that may come out when the servers start to fail unpredictably and also because of the asynchronous network. We will show case-by-case that most of them will be solved by the safety insurance provided by raft.

P1: Follower failure

One failure case we want to handle is when one or more followers are unreachable, regardless of why they are (network partition or the server just fail). In this case, the coordinator will simply retry the AppendEntry request indefinitely, until the follower is back online and reply with confirmation, or if the coordinator simply is replaced by another coordinator election. The rest of the system will not be affected.

Since the logs are persisted into the disks, they will survive the server crashes. When a previously failed server is back online, it should first read the log and apply all the committed entries to restore the state to the latest consensus. The server should also send a request to the coordinator for any log entries that the server may have missed out on while it was offline.

P2: Coordinator failure

The coordinator may fail just like any other server; this will result in a new round of election. However, when coord failure occurs, there’s no guarantee that the log for the previous coordinator is successfully replicated in the majority of the followers.

The coord sends periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no heartbeat from the coord over a period of time called the *election timeout*, then the follower assumes there is no viable leader and begins an election to choose a new leader.

When a new coordinator is elected, it will first need to repair the potential log inconsistency. This is the case we mentioned before where the coordinator can overwrite the logs of the followers if the log in the coordinator is not consistent with the log in the follower. To do so, for each of its followers, the coordinator will broadcast its log to all of the followers. The followers will then find the last entry where they agree, then delete all the entries coming after this critical entry in the follower log and replace them with the coord’s log entries. This includes appending any entries from the coord’s log that the server never received. This mechanism will restore log consistency in a cluster subject to failures.

It’s also possible that the newly elected coordinator has less log entries replicated and therefore the repair process may cause some followers to lose its state even if they are more up to date. To avoid that, we always try to elect the coordinator with the most complete log. Therefore we introduce an *election restriction check*. During elections, choose candidate with log most likely to contain all committed entries Candidates include log info in RequestVote RPCs (index & term of last log entry) voting server V denies vote if its log is “more complete”: (lastTermV > lastTermC) || (lastTermV == lastTermC) && (lastIndexV > lastIndexC). Then the coordinator will have the “most complete” log among the electing majority.

Todo: What happens when the majority of servers fail (together with coord)? → remaining servers elect a new coord, but then coord should realize that the remaining servers do not constitute a majority. (i.e. If 50 servers were in the system and 26 die, 24 remain. The newly elected coord must detect that 24 < 50/2 and thus should broadcast that the system should terminate.

P3: Multiple Elections

To avoid concurrent elections, Raft ventures for a simple solution — randomized timeout. The key idea here is that randomization reduces the probability of election collision. If a split vote happens, we can simply retry with a very large probability of getting it done the next round.

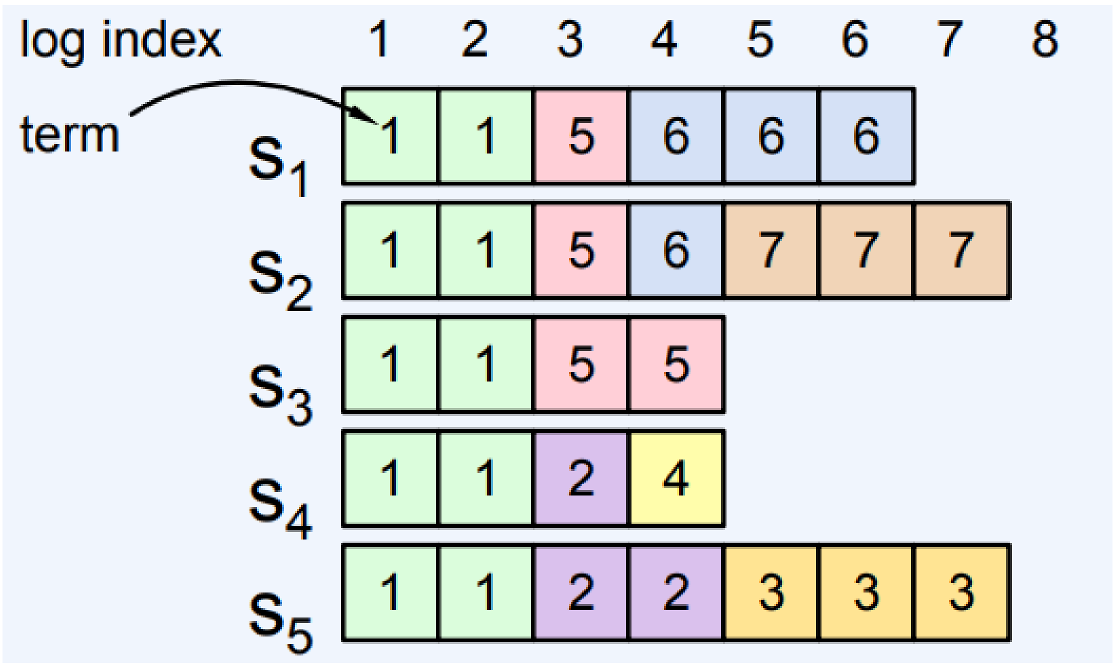

Fig 4

Suppose we have 5 servers and it's shown as fig 4, S4 and S5 won't be able to be elected because their logs have less term number than the majority of the servers.

P4: Multiple Leaders and leader rejoin

Leader in Raft is equivalent to the coordinator in our system. If an old coordinator is cut off, it won't be able to reach the majority. Therefore, even if it identifies itself as a coordinator, it's actually out of the system. The rest of the system will elect a new coordinator to replace it.

When the old coordinator rejoins the network successfully, it will again get the access to reach the majority. However, when a new election is conducted, the candidate increases the Term number by 1. The updated term will be propagated to all nodes who voted for the new coordinator, which are guaranteed to be the majority. Now, if the old coordinator rejoins the group, it'll discover that it has a lower Term number — time to step down. The same principle applied even if the coordinator is always in the system: if a new coordinator is elected and broadcasted with a larger term number, the old coordinator with less term number will be reverted back to a normal follower.

API Design

The following section list the rough API design of our system, they are categorized into Client, Server, Raft:

1. Client:

a. *MakeClient(servers []*rpcutil.ClientEnd) *Client*
b. *(c *Client) Start(key string) notify-channel, err*
c. *(c *Client) Get(key string) err*
d. *(c *Client) Put(key string, value string) err*
~~e. *(c *Client) ServerRejoin(args *ServerJoinArgs, reply *ServerJoinReply) error*~~
~~f. *(c *Client) ServerFail(args *ServerFailArgs, reply *ServerFailReply) error*~~

The API for the client is mostly the same as a3 except with some extra rpc endpoints for the server to notify the client about different situations.
~~*ServerFail* is when the current server that the client is connected to loses its connection with other servers but it's still connected to the client. In this case the server should notify the client about it so that the client will try to connect to a different server.~~

The client will not be notified about the details of raft election or new coordinator; the use of a coordinator is transparent to the client. Instead, the client's requests will be forwarded by the server it is directly connected to (this can be any server in the cluster including the coordinator itself) to the current coordinator. If a new coordinator is elected, then the requests will be forwarded to the new address by the server. All the details are hidden from the client.

2. Server:
    a. *Start(serverId uint8, serverAddr string, serverListenAddr string, serverHeartbeatAddr string) error*
    b. *(kvs *KVServer) Get(args *GetArgs, reply *GetReply) error*
    c. *(kvs *KVServer) Put(args *PutArgs, reply *PutReply) error*
    d. *(kvs *KVServer) NewServerJoin(args *ServerJoinArgs, reply *ServerJoinReply) error: RPC endpoint for server joins. Called by the newly joining server when it notifies this server that it is joining the system.*

3. *Raft:*
    a. *(rf *Raft) GetState() RaftState:* Get the current state of raft, including the leader/coordinator id, term, latest commit index and possibly other info
    b. *(rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) error:* Request vote rpc endpoint, election restrict check should be done here
    c. *(rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply) error:* AppendEntries rpc endpoint, called when the coordinator issue the AppendEntries event
    d. *Start(peers []*rpcutil.ClientEnd, me int, persister *Persister, applyCh chan ApplyMsg) *Raft:* Starting endpoint of the entire raft package

e. *(rf *Raft) NewServerJoin(server *rpcutil.ClientEnd) error*

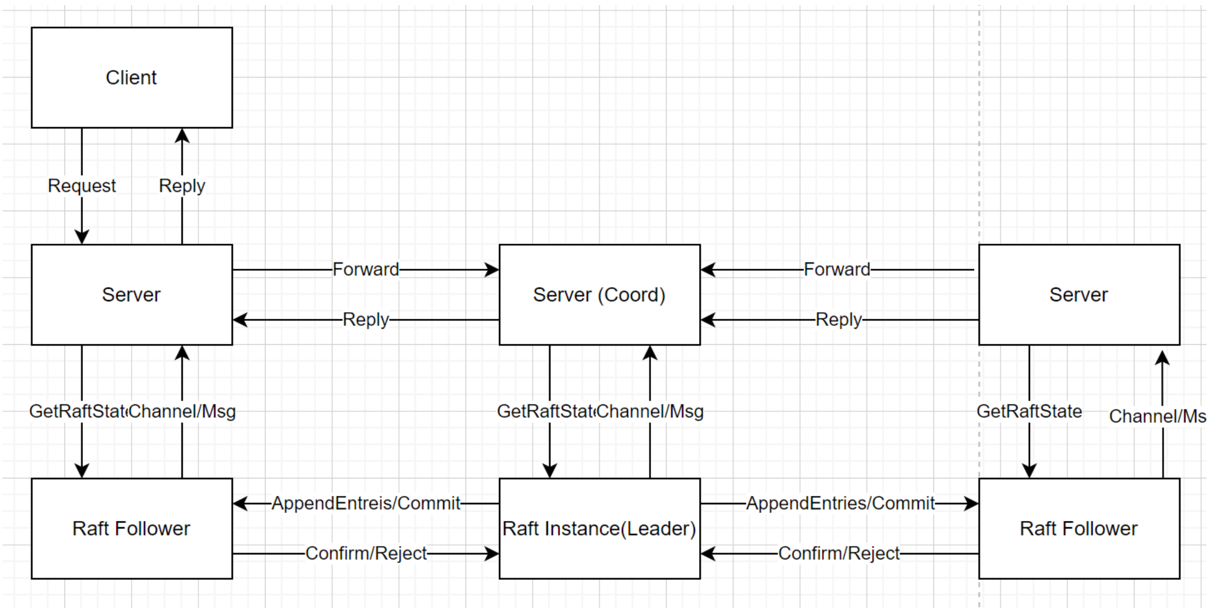

Fig 5 Arch Diagram

Fig 5 shows the rough architecture of our system. The arrows represent the operations one instance can perform by calling the API of another instance. For example, we can imagine each entity of the figure as a struct, and that they communicate with each other by calling the API defined in their structs. The arrows between raft instances are the raft functionality, such as AppendEntries and Commit, which are used to ensure the consistency of the system.
The diagram also shows how, as stated earlier, client requests are forwarded directly from the server contacted by the client to the coord server. The coord communicates to the other servers (raft followers) via its raft instance to ensure consistency, then it returns the response to the server from which it received the request. Finally, the server which initially received the request from the client returns the response to the client.

Timeline

March 18
- Project proposal drafts

March 25
- Final project proposals - after approval, distribute tasks

April 1 (M1)
- Implement the raft mechanism (Jerry, Kehong, Zipeng)
- Implement the stable server node (Andrew, David)
- Implement the client node (Amy)

April 4-8
- Project update meetings

April 8 (M2)

- Integration of raft, server, and client - happy path working

April 15 (M3)
- Implement external server join process
- Test server failure and recovery

April 19
- Project code and final reports

April 18-22
- Project demos

Milestones

- M1: Implement the raft functions which includes propose, voting, appendEntries, commit. Implement a kv server cluster which can communicate with each other and forward requests to the coordinator. Implement client nodes.
- M2: Integration of raft and server-client. Happy path should be working.
- M3: Implement failure recovery (reapply the operations persist in log), new coordinator log repair. External server join (server that's not in the original set).

Evaluation Methodology

- M1: System should work when no failures occur. Gets/Puts should be serviced in some serializable order~~, the system should be able to elect 1st coord and run until N/2 + 1 servers are stopped~~. Failure recovery is not required at this stage. For this milestone, we can assume the coord and other servers will not fail.
    - Traces: Server join (for system initialization), Get, Put

- M2: The raft should be integrated into the servers. The servers will be able to forward the requests to the coordinator and start a new round of raft election when the coordinator dies.
    - Traces: Coord election (initial coord), Raft log traces

- M3: The failure recovery should be implemented. When a server is back online, it should be restored to the state before crashes and further synchronized with the coordinator.
    - Traces: ServerFail, Server join (re-join), CoordFail, ServerFail, Election tracing (for new coords)

Tracing

We will use tracing to help ensure correctness and visualize system executions. Many of these trace actions are similar to those of a3.

- Server tracing (strace)
    - ServerStart{*serverId*}: Recorded in strace. Signifies the start of a server node
    - ServerJoining{*serverId*}: Recorded in strace of the joining server. Indicates that the server with id *serverId* is about to join (or re-join) the system.

    - ServerJoined{*serverId*}: Recorded in strace of the joining server. Indicates that the server with id *serverId* has joined (or re-joined) the system.
    - CoordFail*{serverId}:* Recorded in strace. Indicates that an election timeout has occurred in the server. This will prompt a new election.
- Coord tracing (ctrace)
    - ServerFail{*serverId*}: Recorded in ctrace. Indicates that the coord detected failure of server with id *serverId*.
    - SystemReady*{}*: Recorded in ctrace. Indicates that (1) N servers have joined the system, (2) the first coord server has been elected, and (3) the KVS is ready to process client operations.
- Coord Election tracing (etrace)
    - Proposal*{serverId, term}:* Recorded in etrace. Indicates that an election timeout has occurred, prompting a server to start a new election with itself as a leader. Note that this will be followed by a single VoteConfirm{serverId, serverId, term} since the will vote for itself.
    - VoteRequest*{serverId, term}:* Recorded in etrace. Indicates that a coord candidate has requested a vote for its proposal.
    - VoteConfirm*{serverId, candidateServerId, term}:* Recorded in etrace. Indicates that a server has voted for the proposal {candidateServerId, term}. Must be preceded by a VoteRequest{candidateServerId, term}
    - VoteReject*{serverId, candidateServerId, term}:* Recorded in etrace. Indicates that a server has against the proposal of {candidateServerId, term}. Must be preceded by a VoteRequest{candidateServerId, term}
    - ProposalAccepted*{serverId, term}:* Recorded in etrace. Indicates that a proposal has received a majority (N/2+1) confirmation votes. Must be followed by CoordElected{serverId}
    - ProposalRejected*{serverId, term}:* Recorded in etrace. Indicates that a proposal has not received a majority of confirmation votes.
    - CoordElected*{serverId}*: Recorded in etrace. Indicates that a coord candidate has received a majority of confirmed vote for its proposal, and it has become the new leader.
- Raft log tracing (ltrace)
    - AppendEntry*{term, operation}:* Recorded in ltrace. Indicates that the coord is proposing a new entry to the log
    - AppendConfirm*{term, operation}:* Recorded in ltrace. Indicates that a server approves adding {term, operation} to the log.
    - AppendReject*{term, operation}:* Recorded in ltrace. Indicates that a server rejects adding {term, operation} to the log.
    - CoordCommit*{term, operation}*: Recorded in ltrace. Indicates that a coord has committed {term, operation} to its log. A CoordCommit must be preceded by at least N/2+1 AppendConfirm{term, operation} actions.
    - ServerCommit*{term, operation}:* Recorded in ltrace. Indicates that a server has committed {term, operation} to its log. Every ServerCommit should follow a CoordCommit{term, operation}.
- Client tracing (ktrace)
    - KvslibStart{*clientId*}: Recorded in ktrace. Signifies the start of a client's kvslib. For each unique *ClientId* corresponding to a running client, the *tracing log as*

a whole should contain this action exactly once, as part of ktrace. This action should happen-before all other actions recorded by the client with *ClientId*.

    - KvslibStop{*clientId*}: Recorded in ktrace. For a specific *clientId*, kvslib records this action to mark the end of a client session, corresponding to the completion of KVS.Stop(). For each *ClientId* corresponding to a running client, *the tracing log as a whole* should contain this action exactly once, as part of ktrace. This action should happen-after all other actions reported by a given *clientId*.

- Client operation tracing (get and put traces)
    - PutRecvd{*clientId, key, value*}: Recorded in put trace. Recorded by a server that is connected to the client (at the time of recording this action) and indicates that it has received a put operation from the client.
    - PutFwd{*clientId, key, value*}: Recorded in put trace. Recorded by a server just before it forwards the corresponding put operation to the coord server.
    - PutFwdRecvd{*clientId, key, value*}: Recorded in put trace. Recorded by a coord when it has received the corresponding put operation from a server.
    - PutResult{*clientId, key, value*}: Recorded in put trace. Recorded by the server that replies with a confirmation to the client about the corresponding put operation.
    - GetRecvd{*clientId, key*}: Recorded in get trace. Indicates that the (tail) server has received the corresponding get operation from the client.
    - GetResult{*clientId, key, value*}: Recorded in get trace. Recorded by the (tail) server that replies to the client with result of executing the corresponding get operation

References

1. Anon. The raft consensus algorithm. Retrieved March 25, 2022 from https://raft.github.io/
2. Anon. Designing for understandability: The raft consensus algorithm. Retrieved March 25, 2022 from https://raft.github.io/slides/uiuc2016.pdf
3. Diego Ongaro and John Ousterhout (2014). In Search of an Understandable Consensus Algorithm (Extended Version), June 2014.