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

**Amy Liu, Jerry Liu, Kehong Liu, Zipeng Liang, David Jumao-as-Miller, Andrew Sung**

**Proposal: CPSC 416 RaftKV Final Proposal**

## Introduction and Background

In the project, we present a distributed key value store which uses Raft as backbone support for providing consensus and failure handling. Much of our design is influenced by the Raft extended paper, which details a straightforward and intuitive consensus alternative to Paxos [2]. Our application provides handling for common failures like network partition or node failure and only becomes unresponsive when the majority of all working servers fail. It also handles rejoining when previously failed servers are back online.

## System Design

We wait for the majority of servers to join before clients can send ops. After the initial joining phase, servers can fail and restart. Each server and each client is identified by a unique serverID/clientID, none of the 2 clients/servers share the same ID. The client assigns a local OpId to each request, mainly for the ease of proving ordering correctness. 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 make:

1. There will be N known servers
2. Servers may fail and lose connection
3. Data on disk will not be corrupted or lost
4. The logs will eventually be consistent
5. The system will always have at least ⌊N/2⌋ + 1 servers, i.e. the majority of N servers, up and running in the same partition so that the entire system will never fail
6. Client requests will not arrive at the leader before the leader fully applies its persisted log

### Assumptions we do not 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

### Goals

1. The client is able to send Get/Put requests and get the result
2. The server is able to receive a request issued by client and, if it is not the leader, forward it to the correct leader server
3. The leader server is able to retrieve data from its local data store on a Get, as well as update its local data store on a Put
4. A leader can be successfully elected with a majority vote

5. A leader is able to send log entries through AppendEntries and receive the result for it
6. A leader is able to commit and apply changes to its local data store when it receives successful AppendEntries responses from the majority of Raft nodes
7. Any Raft instance is able to restore its log state after crashing

## Leader as Coordinator

For the simplicity of ordering all the clients' requests and ensuring consistent state, there is a coordinator server that handles all requests. The client is allowed to connect to any arbitrary server, but all requests are ultimately handled by the coordinator since servers simply forward requests to the coordinator. When the coordinator is unreachable, including the case where a coordinator is not yet elected, the server simply drops any incoming requests.

All servers have an in-memory data store (a simple key-value map). On Puts, the coordinator updates its data store after it obtains a consensus on the state change, and only then does it return the response to the caller of the Put request. Get requests are processed by simply looking up and returning the stored value in the coordinator's data store.

The coordinator must be elected by the system. In the initial state of the system, there is no coordinator. All the servers are followers and they hold a timer with a random timeout. When a server has a timeout, it becomes a candidate, resulting in an increase in the current term number and a new election round which is started via broadcasting a vote request to all the other servers as shown in Fig 2 and 3.

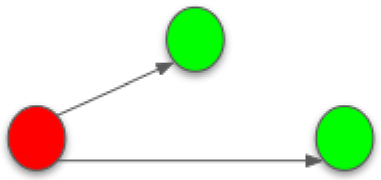


*Fig 2. Timeout and broadcast*

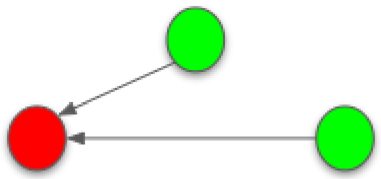


*Fig 3. Receive vote from peers*

If no coordinator is elected in a term, the server that timed out increments the term number and proposes itself as the coordinator again to start a new round of election.

## Local State Machine and Replica

All servers, including the coord, may crash unpredictably. Failure in this system is handled by sharing state among different working nodes so that the state will not be completely lost if only part of the system dies. This is done via log replication.

The Log in our system is for recording the actions the system has reached consensus on. Each log entry contains a log index, the term number, and the operation/request that we share across the servers. The only state-modifying operation in our system is Put, so we do not log Get operations. This log is persisted onto disk at each node for use in failure recovery.

The commit event is then sent to all the followers if the majority of followers reply to the AppendEntreis, and they apply the same entry to their local state so that logs are consistent throughout the entire cluster. The Raft instance applies all the committed entries to the local data state after it commits a log entry.

## Implementation

We successfully implemented the following features, as mentioned in our proposal:
1. Server can process Get/Put requests from the client via forwarding if necessary
2. Client can switch servers when the server it was connected to becomes unreachable
3. Server can handle Get/Put requests and store the key-value state in its local data store
4. System can survive failures and rejoins of any minority of nodes
5. A new leader can be elected when the old leader becomes unreachable

We believe our system in its current state safely supports the aforementioned behaviour. However, Gets at the same key are currently sent synchronously by the client, so we intend to make this asynchronous by the demo date. Otherwise, any changes to the codebase will be minor — mainly bug fixes or non-functional changes for facilitating the demo.

### Client:

The API for the client is the same as a3 although implementation varies slightly. 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. All the details are hidden from the client.

### Server:

The API for the server is similar to a3 but with additional functionality to support Raft integration and a star configuration of nodes (instead of a chain).

### Raft:

Raft instance serves as an internal component of the servers. There are 3 major features:
1. RequestVote: This includes a set of APIs (check the API) to request vote as candidate in raft and process the result.
2. AppendEntries: A set of APIs to ensure that the leader is constantly informing all the followers to reach a consensus state where they share the same logs. It also includes some other functions used to communicate with the server and persist data.
3. Persist/ReadPersist: This feature is for the raft to persist its log and other necessary information into disk. The persisted data is then read and deserialized if the server ever fails and gets restarted.

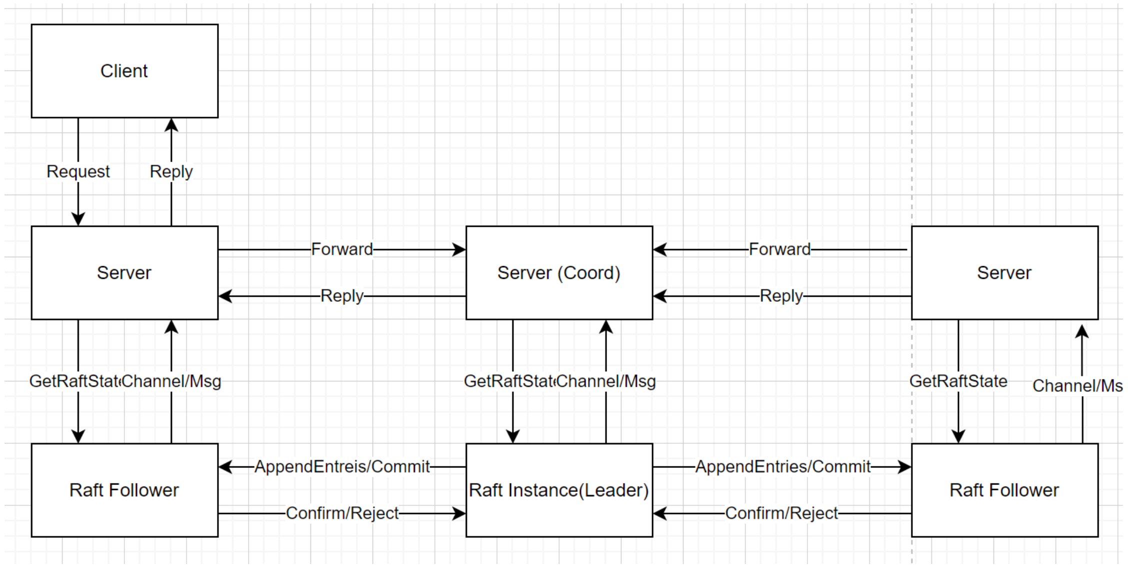


*Fig 4. Arch Diagram*

The diagram above illustrates how requests are processed by the system. Requests from the server contacted by the client are forwarded to the coord server, which then communicates to raft followers (the other servers) before returning a 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.

## Evaluation Methodology

The major evaluation of our system is done by tracing, since this can prove both correctness and order of operations. The Shiviz logs for tracing visualization are placed in our repo under the docs/ folder. Many of these trace actions are similar to those of a3.

- Server tracing (strace)
  - ServerStart{*serverId*}: Signifies the start of a server node. Recorded exactly once in the server's lifetime unless the server fails before the event.
  - ServerListening{*serverId*}: Indicates that the server is now accepting RPCs. If logged, there is exactly one ServerStart logged prior to it in the server's lifetime.
- Raft tracing (rtrace)
  - RaftStart{*id*}
  - Apply {*command, term, index*}
- RequestVote tracing (vtrace)
  - RequestVote{term, id}: Recorded in trace to indicate that a candidate with id is seeking vote from other followers for a given term.
  - RequestVoteRecv{*serverId, term*}*:* Indicates a request vote is received at the follower end for <serverId, term>.
  - RequestVoteRes{*voteGranted, serverId, candidateServerId, term*}*:* Indicates the response of a request vote that is sent from a follower to the leader.
  - LeaderElected{*serverId, term*}*:* Indicates that a leader has been elected via majority vote.
- AppendEntries tracing (atrace)

  - AppendEntries{*term, entries, prevLogIndex, prevLogTerm*}*:* AppendEntries is the main form of communication between raft's instances. It is sent by the leader to followers to reach consensus within their log entries.
  - AppendEntriesRecv{*term, entries, prevLogIndex, prevLogTerm*}*:* Indicates that an AppendEntries call is received by the follower
  - AppendEntriesRes{*valid, conflictIndex*}: Indicates a result for AppendEntries call (including whether it's valid or not) and the index of a log where a conflict happens (if any)
  - Commit{*term, operation*}: Indicates that the leader has received a majority of reply AppendEntries with valid=true. The entry will be committed and applied to the local data store state.
- 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*.
  - NewServerConnection*{clientId, serverId, serverAddr}*: Recorded in ktrace when a client establishes an RPC connection with a server. This action is recorded once in Start() (if the system is available at that time) and every time an operation times out.
- Operation tracing (get and put traces)
  - GetStart{*clientId, opId, key*}/PutStart{*clientId, opId, key, value*}: Recorded by the client just after the user calls Get/Put on the client.
  - GetSend{*clientId, opId, key*}/PutSend{*clientId, opId, key, value*}: Recorded by the client just before sending the Get/Put request to the server. Logged at least once per GetStart/PutStart.
  - GetRecvd{*clientId, opId, key*}/PutRecvd{*clientId, opId, key, value*}: Recorded by the server just after receiving a Get/Put request from the client. Logged at least once per corresponding GetStart/PutStart and at most twice per GetSend/PutSend (1 at follower, 1 at leader).
  - GetFwd{*clientId, opId, key*}/PutFwd{*clientId, opId, key, value}*: Recorded by a follower server just before it forwards the corresponding Get/Put operation to the coord server. Logged at most once per GetRecvd/PutRecvd.
  - RaftPutReq{*cliendId, opId, key, value*}: Recorded on the put trace by the leader's Raft instance just before it appends a Put operation to its log. Logged at least once per PutStart and at most once per PutRecvd.
  - GetResult{*clientId, opId, key, value*}/PutResult{*clientId, opId, key, value*}: Recorded by the leader server, confirming that the corresponding Get/Put result is now on the return path. Logged at least once per GetStart/PutStart and at most once per GetRecvd/PutRecvd.
  - GetResultFwd{*clientId, opId, key, value*}/PutResultFwd{*clientId, opId, key, value*}: Recorded by a follower server just before it forwards a Get/Put result that it received from

the leader back to the client. Logged at least once per GetStart/PutStart and at most once per GetResult/PutResult.

## Demo

We will start the demo by briefly explaining what our project is and what we tried to achieve. We plan to demo the correctness and failure-resilience of our system by showing the terminal output and ShiViz visualization of the traces that we obtain in the following scenarios:

Normal operation

1. Leader election: Start 3/7 servers. No LeaderElected event should be logged at any node. Upon starting the 4th server, one of the servers should eventually log RequestVoteRes with VoteGranted=true returned from the 3 other servers. Only this elected leader should log LeaderElected (until another election round begins).
2. Get/Put correctness and order of operations: Start 4/7 servers and 1 client. Call Put(k1, v1) then Get(k1) on the client. The GetResult should have value v1, and its OpId should be greater than that of the PutResult. Start another client, and let it call Get(k1). This GetResult's value should also be v1.
3. Get/Put correctness and order of operations: Start 4/7 servers and 1 client. Call Get(k2) then Put(k2, v2) on the client. The GetResult should have an empty value, and its OpId should be lower than that of the PutResult.
4. Asynchrony and order of operations: Start 4/7 servers and 1 client. Use a script to call a series of Puts and Gets at the key k1 and one final Get at key k2. The OpId of each result should match the order in which we called the operations, and for each Get at k1, the result should reflect the value of the Put with the highest OpId that is lower than its (the Get's) OpId. The Get at k2 may return before some ops issued before it return.

Surviving 3+ node failures

5. Leader reelection: Start 7/7 servers and wait until a LeaderElected event. Kill 3 servers, one of which is the leader. At least one of the remaining servers should initiate an election round. The node that eventually logs LeaderElected should have logged SendRequestVote for each of the other servers, and it should have received RequestVoteRes with VoteGranted=true from all 3 other servers.
6. Transparency of mid-request node failure: Same setup as 4, but before all requests return, we kill 3 of the servers, some of which lie on the path between the client and the leader. The returned OpIds and values should be identical to the case where no nodes failed. The client should log at least one NewServerConnection event. We can try this with both a failing follower and a failing leader, and we expect the same results.

Utilizing 3+ new nodes

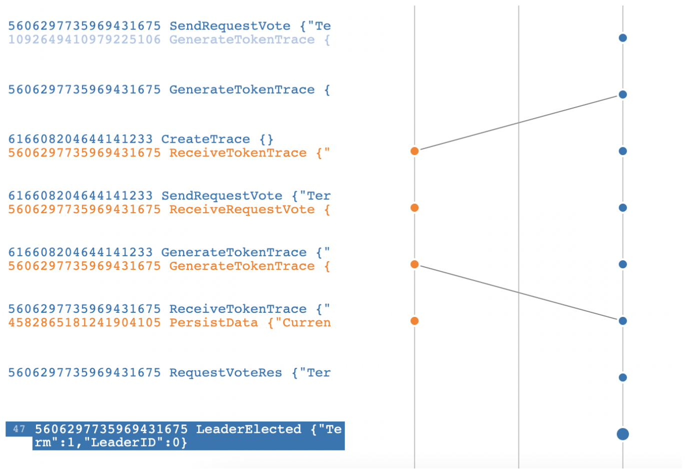
7. State syncing: Start 4/7 servers and 1 client. Once a leader is elected, call Put(k1, v1) and wait for its result. Start the remaining 3 servers, and connect a client to one of the new servers. Call Get(k1) on the new client. The result of the Get should be v1.
8. State syncing from new leader: Start 4/7 servers and 1 client. Once a leader is elected, call Put(k1, v1) and wait for its result. Add a server, kill the leader, wait for reelection to end, then

add the remaining servers. Connect a client to the last server added. The GetResult's value should still be v1.

9. Transparency of mid-request node failure and recovery: Same setup as 6, but restart the servers immediately after failing them. From the user, the observed events and results should match scenario 6.

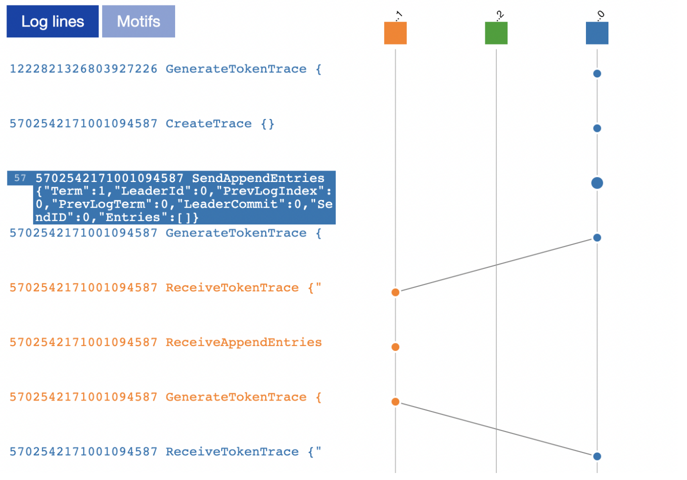
## ShiViz Visualizations

### raft0 requests votes and becomes the leader



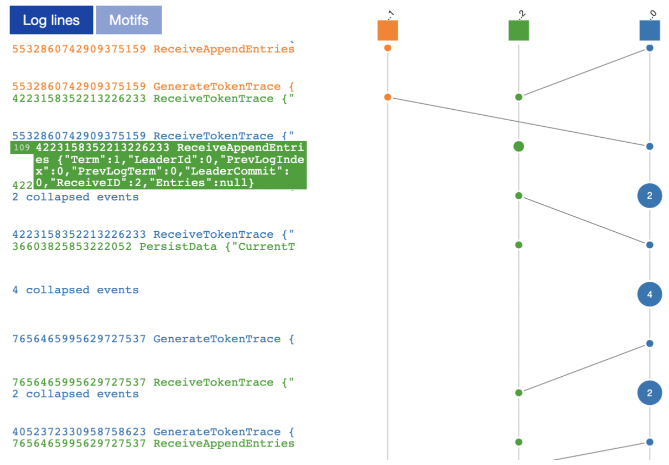
```
5606297735969431675 SendRequestVote {"Te
1092649410979225106 GenerateTokenTrace {

5606297735969431675 GenerateTokenTrace {

616608204644141233 CreateTrace {}
5606297735969431675 ReceiveTokenTrace {"

616608204644141233 SendRequestVote {"Ter
5606297735969431675 ReceiveRequestVote {

616608204644141233 GenerateTokenTrace {"
5606297735969431675 GenerateTokenTrace {

5606297735969431675 ReceiveTokenTrace {"
4582865181241904105 PersistData {"Curren

5606297735969431675 RequestVoteRes {"Ter

47 5606297735969431675 LeaderElected {"Te
   rm":1,"LeaderID":0}
```

### raft0 sends appendentries to raft1 and receives reply



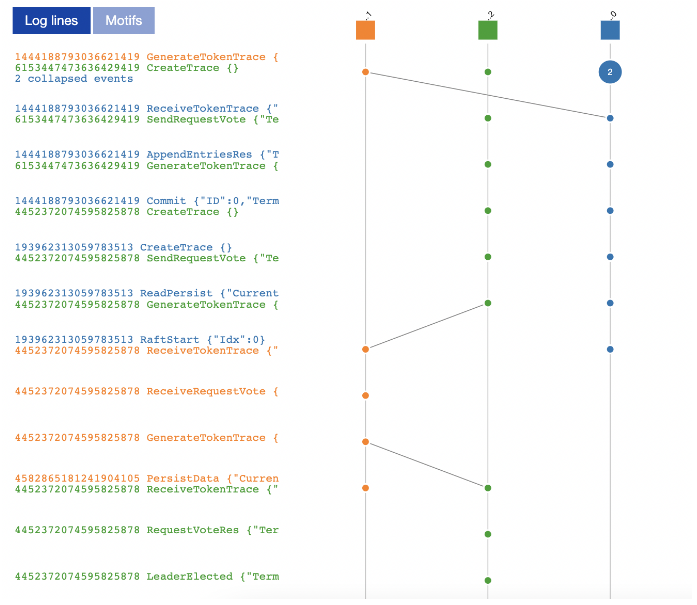
```
1222821326803927226 GenerateTokenTrace {

5702542171001094587 CreateTrace {}

57 5702542171001094587 SendAppendEntries
   {"Term":1,"LeaderId":0,"PrevLogIndex":
   0,"PrevLogTerm":0,"LeaderCommit":0,"Se
   ndID":0,"Entries":[]}
5702542171001094587 GenerateTokenTrace {

5702542171001094587 ReceiveTokenTrace {"

5702542171001094587 ReceiveAppendEntries

5702542171001094587 GenerateTokenTrace {

5702542171001094587 ReceiveTokenTrace {"
```

raft2 joins the system as a follower



```
5532860742909375159 ReceiveAppendEntries

5532860742909375159 GenerateTokenTrace {
4223158352213226233 ReceiveTokenTrace {"

5532860742909375159 ReceiveTokenTrace {"
109 4223158352213226233 ReceiveAppendEntri
    es {"Term":1,"LeaderId":0,"PrevLogInde
    x":0,"PrevLogTerm":0,"LeaderCommit":
422 0,"ReceiveID":2,"Entries":null}
2 collapsed events

4223158352213226233 ReceiveTokenTrace {"
36603825853222052 PersistData {"CurrentT

4 collapsed events

7656465995629727537 GenerateTokenTrace {

7656465995629727537 ReceiveTokenTrace {"
2 collapsed events

4052372330958758623 GenerateTokenTrace {
7656465995629727537 ReceiveAppendEntries
```

raft0 restarts; raft2 requests votes and becomes the leader



```
1444188793036621419 GenerateTokenTrace {
6153447473636429419 CreateTrace {}
2 collapsed events

1444188793036621419 ReceiveTokenTrace {"
6153447473636429419 SendRequestVote {"Te

1444188793036621419 AppendEntriesRes {"T
6153447473636429419 GenerateTokenTrace {

1444188793036621419 Commit {"ID":0,"Term
4452372074595825878 CreateTrace {}

193962313059783513 CreateTrace {}
4452372074595825878 SendRequestVote {"Te

193962313059783513 ReadPersist {"Current
4452372074595825878 GenerateTokenTrace {

193962313059783513 RaftStart {"Idx":0}
4452372074595825878 ReceiveTokenTrace {"

4452372074595825878 ReceiveRequestVote {

4452372074595825878 GenerateTokenTrace {

4582865181241904105 PersistData {"Curren
4452372074595825878 ReceiveTokenTrace {"

4452372074595825878 RequestVoteRes {"Ter

4452372074595825878 LeaderElected {"Term
```

## 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.
4. MIT 6.824 Lab Fall 2022: https://pdos.csail.mit.edu/6.824/labs/lab-raft.html
5. MIT 6.824 Lab Repo: https://github.com/WenbinZhu/mit-6.824-labs/tree/c996f20a5de58b27b5ee5e1b13e90e4aeb50d745