TLA is quite **a concrete version of implementation**, but may not be used as simulation directly.

States $\overset{decide}{\longleftrightarrow}$ Messages

States

TLA	Simulation
log	✓*
currentTerm	✓
state	✓*
votedFor	✓
commitIndex	✓*
votesResponded	
votesGranted	✓
nextIndex	
matchIndex	✓*

*:The updated ways are different from that in the simulation.

log: The AppendEntriesHandler is different.

state: The simulation allows a leader => follower => candidate without receiving any external message. While the TLA doesn't allow leader => follower.

commitIndex: The AppendEntriesHandler, AppendEntriesResponseHandler are different.

matchIndex: The AppendEntriesResponseHandler is different.

Messages

TLA	Simulation
Client Request	Client Request *(But support interpret Get, Put... for KV operations)*
Append Entries Request *(up to 1 entry)*	Append Entries/Heartbeat Request *(support multiple entries)*
Append Entries Request Resp *(up to 1 entry)*	Append Entries/Heartbeat Request Resp *(support multiple entries)*
Request Vote Request	Request Vote Request
Request Vote Response	Request Vote Response
-	Read Index Request
-	Read Index Response

Simulation

Minimum States and Messages Set

The goal is to ensure the **core states (which decide the KV store)** are correct. The messages will affect the states.
Considering the requests except RO, the **core states** in a server are: **log, commitIndex**. Other states in the simulation like **currentTerm, state, votedFor, votesGranted, matchIndex** will affect **log** or **commitIndex**, and thus we should check them in the simulation.

Similarly, all the messages that affect the above states should be checked, e.g. AppendEntries, RequestVote. **Not all the parameters in these messages should be checked.** Although the **nextIndex** is attached in **AppendEntriesRequest** msg, but it doesn't affect the states.

We do not need to simulate other messages which don't affect the above states. E.g., **PreVote** messages in ETCD doesn't change any of the above states.

Valid States/Messages Set

Suppose a raft server is a state machine, given a sequence of input messages, its valid local states could be different (a valid set).

Sequence	Pre Log State	Post Valid Log state
Client Req Q1, Q2, Q3	||	|Q1|Q2|Q3|,|Q2|Q1|Q3|,|Q1|Q3|Q2|,...
Client Req Q1, Q2, Q3, AppendEntryReq(PreLogIndex:0, Q1)	||	|Q1|Q2|Q3|, |Q1|Q3|Q2|

Sequence	Pre CommitIndex State	Post CommitIndex state
AppendEntryResp(matchIndex: 5)	0	<= 0
AppendEntryResp(matchIndex: 5), AppendEntryResp(matchIndex: 5)	0	<= 5

ETCD allows the following state change: A leader can step down to a follower without receiving any external messages. Such transition is valid.

Sequence	Pre State State	Post State state
nil	Leader	Leader, Follower

The simulation should be **generalizable enough to allow for checking more valid states/transitions/messages**. E.g. the commitIndex should be **as large as possible**.

How to Check?

We still check the validity of the sent out messages.

What is a valid message?

The current valid states set decide the total set of valid sent out messages.

Example:

Valid States Set	Valid Message	Invalid Message
CommitIndex <= 5	AppendEntryReq(CommitIndex: 4)	AppendEntryReq(CommitIndex: 6)

A valid state set S could be like this:
S: set of $State$ {

- self, ID (fixed)
- peers, array of ID (fixed)
- state, State
- currentTerm, Term
- votedFor, ID
- votesGranted, array of bool
- commitIndex, Index
- matchIndex, array of Index
- log, Log
  }

S is a hypothetical set whose size might be infinitely large. For example, suppose current valid commitIndex set are <= 5, then there should be 6 different members in the set: {0, 1, 2, 3, 4, 5}.

Given the valid state set S, the set of valid messages:

- RequestVoteReq(From, To, mterm, mlastLogIndex, mlastLogTerm)
  $\exists s \in S$,
  - Parameters checker:
    - mterm == s.currentTerm
    - (mlastLogIndex, mlastLogTerm) == s.log.lastEntry()
  - State checker:
    - Candidate == s.state
- RequestVoteResp(From, To, mterm, mvoteGranted)
  $\exists s \in S$,
  - Parameters checker:
    - mterm == s.currentTerm
    - !mvoteGranted || To == s.votedFor
  - State checker:
    - Follower == s.state
- AppendEntryReq(From, To, mterm, mprevLogIndex, mprevLogTerm, mentries, mcommitIndex)
  $\exists s \in S$,
  - Parameters checker:
    - mterm == s.currentTerm
    - logMatch(s.log, mprevLogIndex, mprevLogTerm, mentries)
    - mcommitIndex == s.state
  - State checker:
    - Leader == s.state
- AppendEntryResp(From, To, mterm, msuccess, **mmatchIndex**)
  mmatchIndex cannot be inferred from the existing states, but can be solved by adding an additional data "largestMatchIndex" or reusing the matchIndex array in the follower side, to indicate in this term, the largest possible matchindex so far, since in each term, a leader shouldn't propose two different entries in a same slot. (Similar optimization could be applied to commitIndex.)
  $\exists s \in S$,
  - Parameters checker:
    - mterm == s.currentTerm
    - !msuccess || mmatchIndex <= s.largestMatchIndex
  - State checker:
    - Follower == s.state

How to maintain S?

Events

S are updated based on the **events**. A **event** could be:

1. Receive a message.

Event	Pre S	Post S
Receive AppendEntryResp(From: 0, msuccess: true, matchIndex: 5)	{matchIndex: {0, 5, 0}, commitIndex: 0}	{matchIndex:{0, 5, 0}, commitIndex: 0}, {matchIndex:{5, 5, 0}, commitIndex: 0}, {matchIndex:{5, 5, 0}, commitIndex: 1}, ...

2. Send out a message.

Event	Pre S	Post S
Send AppendEntryReq(commitIndex: 5)	{matchIndex:{0, 5, 0}, commitIndex: 0}, {matchIndex:{5, 5, 0}, commitIndex: 0}, {matchIndex:{5, 5, 0}, commitIndex: 1}, ...	{matchIndex: {5, 5, 0}, commitIndex: 5}

3. Nil.

3.1 A leader may step down to a follower if it failed to connect to the quorum:

Event	Pre S	Post S
nil	{state: Leader, term: n}	{state: Leader, term: n}, {state: Follower, term: n}

3.2 A follower will become candidate if timeout:

Event	Pre S	Post S
nil	{state: Follower, term: n}	{state: Follower, term: n}, {state: Candidate, term: n+1}

3.3 A candidate will increment its term itself:

Event	Pre S	Post S
nil	{state: Candidate, term: n}	{state: Candidate, term: n}, {state: Candidate, term: n+1}

3.4 A candidate becomes leader if receives quotum votes:

Event	Pre S	Post S
nil	{state: Candidate}	{state: Candidate}, *{state: Leader}, only apply when votesGranted satisfies the quorum*

Transitions

Specifically, S will be **expanded** if it **receives a message m**:
$S' = S$
for $s \in S'$:
$S = S \cup (s.handle(m))$ (Q1: why keep all the states in the old S?)

and will be **narrowed** if it **sends out a message m**.
$S' = S, S = \emptyset$
for $s \in S'$:
$S = S \cup (s.match(m))$

Since there are some transitions that don't need any events, we need to apply these transitions to S whatever receive/send the messages, for hypothetically infinitely times.

E.g.

Event	Pre S
Receive RequestVoteResp{From: 1, mvoteGranted: true}	{state: Candidate, currentTerm: n, votesGranted: {true, false, false}}

The Post S should be:

- {state: Candidate, currentTerm: n, votesGranted: {true, false, false}} *#the old S*
- $\cup$ {{state: Leader, currentTerm: n, votesGranted: {true, true, false}}} *#case 1: apply the message*
- $\cup$ {{state: Follower, currentTerm: n, votesGranted: {false, false, false}}} *#case 2: case 1 then apply nil transition 3.1*
- $\cup$ {{state: Candidate, currentTerm: n+1, votesGranted: {false, false, false}}} *#case 3: case 2 then apply nil transition 3.2*
- $\cup$ {{state: Candidate, currentTerm: >n+1, votesGranted: {false, false, false}}} *#case 4: case 3 then apply nil transition 3.3 for infinite times*

ETCD examples

Case 1

Q1: why keep all the states in the old S?
In ETCD, sometimes it will just *ignores* the incoming messages: https://github.com/etcd-io/etcd/blob/main/raft/raft.go#L853-L863

To fix this, current simulation will only handle the request when checking the reply:

```
valid = s.RequestVoteReplyChecker(msg_itp.(RequestVoteReply), s.HandleRaftReqMessages(req.(raftpb.Mess
```

Then, in this case, the simulation will not actually handle the request. Such fix assumes that if it doesn't send out the reply, there shouldn't be any side effect to the status.

Considering S, the case should be:

Event	Pre S	Post S
Receive RequestVoteReq(mterm: 100)	{state: Follower, term: 10}	{state: Follower, term: 10}, {state: Follower, term: 100}

Keeping the old S allows ignoring messages.

Case 2

Step to Leader/Candidate.
At first, the simulation is implemented like this:

```
func (s *Simulation) RequestVoteReplyHandler(reply RequestVoteReply) {
        ...

        if s.votesGranted satisfies Quorum {
                s.becomeLeader()
        }
        return
}
```

However this results in false positive considering the following sequence:
Assume there are three servers: 0, 1, 2

1. 0 sends RequestVoteReq to 1
2. 1 replys true to 0 (0 becomes Leader in simulation)
3. 0 sends RequestVoteReq to 2 (check fails since 0 is not Candidate now)

The simulation fixes this issue by moving s.becomeLeader to the sending AppendEntriesReq check part:

```
func (s *Simulation) AppendEntryRequestChecker(args AppendEntriesArgs) bool {
        if s.State != Leader {
                if !s.testLeader() {
                        return false
                }
                s.becomeLeader()
        }
  ...
}
```

Considering S, the case should be:
Assume there are three servers: 0, 1, 2

1. 0 sends RequestVoteReq to 1
2. 1 replys true to 0 (0 becomes Leader in simulation)

Event	Pre S	Post S
Receive RequestVoteResp(From: 1, mvoteGranted: true)	{state: Candidate, votesGranted: {true, false, false}}	{state: Candidate, votesGranted: {true, false, false}}, **{state: Candidate, votesGranted: {true, true, false}}**, {state: Leader, votesGranted: {true, true, false}}

3. 0 sends RequestVoteReq to 2

Event	Pre S	Post S
Send RequestVoteReq()	{state: Candidate, votesGranted: {true, false, false}}, **{state: Candidate, votesGranted: {true, true, false}}**, {state: Leader, votesGranted: {true, true, false}}	{state: Candidate, votesGranted: {true, false, false}}, **{state: Candidate, votesGranted: {true, true, false}}**