1 Introduction

Raft is a consensus algorithm designed for understandability and ease of implementation compared to the Paxos algorithm, and thus, it is widely utilized in various products to resolve consensus issues.

In distributed systems, Raft requires odd number of servers to always maintain a quorum that remains functional and communicative in case of network partition faults. Practically, this means a Raft cluster size should be at least three to tolerate a single point of failure.

While the requirement for an odd cluster size isn't problematic for large-scale distributed systems, it can pose a challenge for budget-constrained customers who need to reach consensus with fewer servers. One solution to this issue is to substitute one server in the cluster with a cost-effective entity known as a witness. The witness acts as a tie-breaker, ensuring the maintenance of a quorum in case of an even split of servers due to network issues. Witness typically operates on low-configuration hardware to save costs, seldom participates in critical data paths, and only persists a minimal amount of data. In the Paxos algorithm, a witness can be implemented as shared storage, which is not only inexpensive but also widely available to most customers.

Efforts have been undertaken to incorporate witness into Raft algorithm. For example, Diego Ongaro's Ph.D. dissertation explored the reduction of server numbers by using a witness to store log entries for any failed server until recovery or replacement. Jehan-François Pâris and Darrell D. E. Long proposed a type of follower-only server witness that doesn't persist user data. TiKV implements a witness in a similar manner as described by Ongaro: a log-only entity that persists only raft logs without applying them, and supports switching between witness and non-witness servers.

However, all existing research and implementations necessitate a standalone server as a witness, adding to deployment complexity and reducing cost-efficiency compared to shared storage in the Paxos algorithm. Additionally, the witness must participate in log replication when it needs to conform to a quorum with other servers (when some servers are down), implying that its low-configuration hardware could potentially become a performance bottleneck in such scenarios.

2 Algorithm

In this section, we present the extended Raft algorithm, which is a variant of the Raft algorithm. It is designed for cluster with regular servers and at most one witness. The witness functions strictly as a follower server, implying that its role never transitions to candidate or leader. Furthermore, a witness only maintains a minimal set of metadata.

Our newly proposed algorithm minimizes data traffic and reduces the frequency of witness visits while preserving all key properties of the Raft algorithm. This implies that the extended Raft algorithm ensures that each of the following properties is consistently upheld.

- Election Safety
- Leader Append-only
- Log Matching
- Leader Completeness
- State Machine Safety

In the rest of this document, unless otherwise specified, the term 'server' will denote either a regular server or a witness. Consequently, the server set, denoted as *Server*, that constitutes a cluster could either be $\mathit{RegularServer} \cup \mathit{witness}$ or simply *RegularServer*, where *RegularServer* is the set of all regular servers.

2.1 Concepts and Definitions

Definition 1. *A leader's* ***replication set*** *is a subset of the cluster server set, with its cardinality matching that of the regular server set.*

Let *ReplicationSets* represent the set of all replication sets within a cluster. We have:

$$\mathit{ReplicationSets} \triangleq \begin{cases} \{\mathit{Server}\}, & \mathit{witness} \notin \mathit{Server} \\ \{\mathit{Server} \setminus \{x\} : x \in \mathit{Server}\}, & \mathit{witness} \in \mathit{Server} \end{cases}$$

A leader maintains its replication set based on its view of the servers and modifies it under specific conditions. A leader may alter its replication set multiple times within a single term. To uniquely identify each replication set, we introduce the notion of a 'subterm' as detailed below.

Definition 2. *A term is segmented into* SUBTERMS*, each begins with a replication set.*

Subterms are sequentially numbered using consecutive integers, beginning from 0 for each term. The leader retains its current subterm and increments it when the replication set is altered. Consequently, a leader's replication set remains static throughout its subterm.

2.2 States

Figure 2.1 illustrates the additional states introduced by the Extended Raft algorithm to accommodate the witness.

Each leader maintains three additional volatile variables: *replicationSet*, *currentSubterm*, and *witnessSubterm*. The *replicationSet* represents the leader's current replication set, initialized to *RegularServer*, and is modified by *AdjustReplicationSet* action. The *currentSubterm* represents the latest subterm of the leader, initialized to 0 and incremented upon a change in the leader's replication set. The *witnessSubterm* denotes the latest subterm during which the leader replicated its log entry to the witness.

In addition to *currentTerm* and *votedFor*, the witness also includes *witnessReplicationSet*, *witnessLastLogTerm*, and *witnessLastLogSubterm*. These represent the most recent replication set, log entry term, and log entry subterm that the leader sent to the witness. We will further discuss this in Section 2.3.

Beyond the additional states in the leader and witness, each log entry is also associated with the leader's current subterm number when it is appended to the leader's log (Figure 2.2). This subterm number is replicated and persisted alongside the log entry on all regular servers. We now denote a log entry as $\langle index, term, subterm \rangle$, which is uniquely identified by *index* and *term*, and associated with *subterm*.

2.3 Log Replication

The leader replicates its log to regular servers in the exact same way as that in the Raft algorithm, and the regular servers handle the received log entries in the same way as well.

However, for the witness, log replication is performed differently, as shown in Figure **??**. In the Extended Raft algorithm, the leader sends an *AppendEntriesToWitnessRequest*

The following variables are used only on leaders:

The next entry to send to each follower.

VARIABLE $nextIndex$

The latest entry that each follower has acknowledged is the same as the leader's. This is used to calculate commitIndex on the leader.

VARIABLE $matchIndex$

Leader's replication set

VARIABLE $replicationSet$

Leader's subterm number

VARIABLE $currentSubterm$

The latest subterm that this leader has written to the witness

VARIABLE $witnessSubterm$

$$leaderVars \triangleq \langle nextIndex, matchIndex, elections, replicationSet, currentSubterm, witnessSubterm \rangle$$

The following variables are persisted only on witness:

The latest replication set sent from leader

VARIABLE $witnessReplicationSet$

The latest term of replicated log entry

VARIABLE $witnessLastLogTerm$

The latest subterm of replicated log entry

VARIABLE $witnessLastLogSubterm$

$$witnessVars \triangleq \langle witnessReplicationSet, witnessLastLogTerm, witnessLastLogSubterm \rangle$$

Figure 2.1: State

Leader i receives a client request to add v to the log.

$$
\begin{array}{l}
\mathit{ClientRequest}(i, v) \triangleq \\
\quad \land\ \mathit{state}[i] = \mathit{Leader} \\
\quad \land\ \text{LET } \mathit{entry} \triangleq [\mathit{term} \mapsto \mathit{currentTerm}[i], \\
\qquad\qquad\qquad\qquad \mathit{subterm} \mapsto \mathit{currentSubterm}[i], \\
\qquad\qquad\qquad\qquad \mathit{value} \mapsto v] \\
\qquad\qquad \mathit{newLog} \triangleq \mathit{Append}(\mathit{log}[i], \mathit{entry}) \\
\qquad \text{IN } \mathit{log}' = [\mathit{log} \text{ EXCEPT } ![i] = \mathit{newLog}] \\
\quad \land\ \text{UNCHANGED } \langle \mathit{messages}, \mathit{serverVars}, \mathit{candidateVars}, \\
\qquad\qquad\qquad \mathit{leaderVars}, \mathit{commitIndex}, \mathit{witnessVars} \rangle
\end{array}
$$

Figure 2.2: Client request

to the witness, which includes the current replication and metadata of the log entry that satisfies the following conditions:

1. The log entry's term and subterm are equal to *currentTerm* and *currentSubterm*, respectively.

2. The leader's current replication set includes the witness.

3. The leader has received acknowledgments for the log entry from at least a subquorum (one server away from forming a quorum) in the leader's current replication set.

4. The leader has not received any acknowledgment from the witness during the current subterm.

The *AppendEntriesToWitness* can be considered as a special implementation of the *AppendEntries* action with batching of log prefix up to a specific log entry. This log entry belongs to the current subterm and has received subquorum acknowledgements from the current replication set. In addition to the message fields that also exist in *AppendEntriesRequest*, the dispatched message *AppendEntriesToWitnessReques* also contains the current replication set and the metadata of the last batched entries.

Upon receiving the request, the witness persists the replication set, as well as the term and subterm of the log entry, then responds to the leader with an *AppendEntriesResponse* message (Figure 2.4).

```
Leader i sends witness an AppendEntriesToWitness request with log entry
in current subterm acknowledged by a subquorum in current replication set.
AppendEntriesToWitness(i) ≜
    ∧ state[i] = Leader
    ∧ WitnessID ∈ replicationSet[i]
    ∧ LET Agree(index) ≜
              {i} ∪ {k ∈ replicationSet[i] : matchIndex[i][k] ≥ index}
          IsAgreed(k)  ≜
              ∧ log[i][k].term    = currentTerm[i]
              ∧ log[i][k].subterm = currentSubterm[i]
              ∧ ({WitnessID} ∪ Agree(k)) ∈ Quorum
          agreeIndexes ≜ {k ∈ 1 .. Len(log[i]) : IsAgreed(k)}
          lastEntry    ≜ Max(agreedIndex)
      IN  ∧ agreeIndexes ≠ {}
          ∧ ∨ witnessSubterm never exceeds currentSubterm following specification
              ∧ currentSubterm[i] > witnessSubterm
              ∧ Send([mtype          ↦ AppendEntriesToWitnessRequest,
                      mterm          ↦ currentTerm[i],
                      mlogTerm       ↦ log[i][lastEntry].term,
                      mlogSubterm    ↦ log[i][lastEntry].subterm,
                      mreplicationSet ↦ replicaitonSet[i],
                      mindex         ↦ lastEntry,
                       mlog and mentries are used as history variable for
                       the proof. They do not exist in a real implementation.
                      mlog           ↦ log[i],
                      mentries       ↦ SubSeq(log[i], 1, lastEntry),
                      msource        ↦ i,
                      mdest          ↦ WitnessID])
              ∧ UNCHANGED ⟨leaderVars⟩
            ∨ shortcut replication
              ∧ currentSubterm[i] = witnessSubterm[i]
              ∧ Send([mtype          ↦ AppendEntriesResponse,
                      mterm          ↦ currentTerm[i],
                      msuccess       ↦ TRUE,
                      mmatchIndex    ↦ lastEntry,
                      msource        ↦ WitnessID,
                      mdest          ↦ i])
    ∧ UNCHANGED ⟨serverVars, candidateVars, logVars, witnessVars⟩
```


Figure 2.3: AppendEntriesToWitness

Witness receives an AppendEntriesToWitnessRequest from server j with m.mterm ¡= currentTerm[WitnessID].

$$
\begin{array}{l}
\mathit{HandleAppendEntriesToWitnessRequest}(j,\ m) \triangleq \\
\quad \land\ m.mterm \leq \mathit{currentTerm}[\mathit{WitnessID}] \\
\quad \land\ \lor\ \text{reject request} \\
\qquad\quad \land\ m.mterm < \mathit{currentTerm}[\mathit{WitnessID}] \\
\qquad\quad \land\ \mathit{Reply}([\begin{array}[t]{ll} \mathit{mtype} & \mapsto \mathit{AppendEntriesResponse}, \\ \mathit{mterm} & \mapsto \mathit{currentTerm}[\mathit{WitnessID}], \\ \mathit{msuccess} & \mapsto \text{FALSE}, \\ \mathit{mmatchIndex} & \mapsto 0, \\ \mathit{msource} & \mapsto \mathit{WitnessID}, \\ \mathit{mdest} & \mapsto j], \\ m) & \end{array} \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle \mathit{serverVars},\ \mathit{logVars},\ \mathit{witnessVars} \rangle \\
\qquad \lor\ \text{accept request, always no conflict.} \\
\qquad\quad \land\ m.mterm = \mathit{currentTerm}[\mathit{WitnessID}] \\
\qquad\quad \land\ \lor\ m.mlastLogTerm > \mathit{witnessLastLogTerm}[\mathit{WitnessID}] \\
\qquad\qquad \lor\ \land\ m.mlastLogTerm = \mathit{witnessLastLogTerm}[\mathit{WitnessID}] \\
\qquad\qquad\quad \land\ m.mlastLogSubterm > \mathit{witnessLastLogSubterm}[\mathit{WitnessID}] \\
\qquad\quad \land\ \mathit{witnessReplicationSet}' = \\
\qquad\qquad [\mathit{witnessReplicationSet}\ \text{EXCEPT}\ ![\mathit{WitnessID}] = m.mreplicationSet] \\
\qquad\quad \land\ \mathit{witnessLastLogTerm}' = \\
\qquad\qquad [\mathit{witnessLastLogTerm}\ \text{EXCEPT}\ ![\mathit{WitnessID}] = m.mlastLogTerm] \\
\qquad\quad \land\ \mathit{witnessLastLogSubterm}' = \\
\qquad\qquad [\mathit{witnessLastLogSubterm}\ \text{EXCEPT}\ ![\mathit{WitnessID}] = m.mlastLogSubterm] \\
\qquad\quad \text{log will not be modified in real implementation. It is only} \\
\qquad\quad \text{used here for the proof.} \\
\qquad\quad \land\ \mathit{log}' = [\mathit{log}\ \text{EXCEPT}\ ![\mathit{WitnessID}] = m.mentries] \\
\qquad\quad \land\ \mathit{Reply}([\begin{array}[t]{ll} \mathit{mtype} & \mapsto \mathit{AppendEntriesResponse}, \\ \mathit{mterm} & \mapsto \mathit{currentTerm}[\mathit{WitnessID}], \\ \mathit{msuccess} & \mapsto \text{TRUE}, \\ \mathit{mmatchIndex} & \mapsto m.mindex, \\ \mathit{msource} & \mapsto \mathit{WitnessID}, \\ \mathit{mdest} & \mapsto j], \\ m) & \end{array} \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle \mathit{serverVars},\ \mathit{log} \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{candidateVars},\ \mathit{leaderVars} \rangle
\end{array}
$$

Figure 2.4: Handle AppendEntriesToWitness

Note that the *mlog* and *mentries* fields in the message are solely used for proof. They do not exist in the actual implementation, nor will the witness persist the full log prefix in its durable storage. The persistence of *m.mentries* to *log* in *HandleAppendEntriesToWitnessRequest* is exclusively used for proof purposes.

When the leader receives an acknowledgment from the witness, it also updates its *witnessSubterm* to the current subterm if the acknowledged log entry is in the current subterm. Then, according to condition **??**, the leader no longer sends *AppendEntriesToWitnessRequest* in the current subterm. Instead, the leader treats subsequent log entries as already acknowledged by the witness if they fulfill all conditions except condition **??**. This is referred to as **Shortcut Replication**, where the leader sends an *AppendEntriesResponse* to itself on behalf of the witness.

Note that the log entry in *AppendEntriesToWitness* must be within the current term and subterm. So, when the leader advances its subterm (a consequence of replication set adjustment), a new empty log entry must be appended to the leader's log. This is to ensure the leader won't be impeded by condition 1 in committing log entries.

Leader immediately commits a log entry during its term when the log entry is acknowledged by a quorum, in the same way as in Raft. The acknowledgment either comes from a regular server or the witness. Since the leader sends *AppendEntriesToWitnessRequest* to the witness only after it has received subquorum acknowledgments from regular servers, any log entry acknowledged by the witness is immediately committed.

2.4 Replication Set Adjustment

Leader maintains the replication set and modifies it whenever necessary. This process is referred to as **Replication Set Adjustment**. Figure 2.5 illustrates a straightforward method to adjust the replication set. Although there could be multiple ways to adjust a replication set in practical implementations, such a simple method can be used in the algorithm to minimize the atomic regions without loss of generality.

To commit, the leader's replication set must contain at least a subquorum of healthy regular servers if it includes a witness. Therefore, in a practical implementation, replication set adjustment should be performed in a way such that the resulting replication set includes as many healthy regular servers as

Adjust replication set on leader i. This action updates replication set by swapping items inside and outside. While implementations may change replication set in various ways, the spec uses this simple swapping to minimize atomic regions without loss of generality.

$$
\begin{aligned}
&AdjustReplicationSet(i) \triangleq \\
&\quad \land\ state[i] = Leader \\
&\quad \land\ replicationSet[i] \neq Server
\end{aligned}
$$

Swap server outside replication set with some server inside

$$
\begin{aligned}
&\quad \land\ \text{LET } in \triangleq \text{CHOOSE } x \in replicationSet[i] : x \neq i \\
&\qquad\qquad out \triangleq \text{CHOOSE } x \in Server \setminus replicationSet[i] : \text{TRUE} \\
&\qquad \text{IN} \quad replicationSet' = \\
&\qquad\qquad [replicationSet \text{ EXCEPT } ![i] = (@ \setminus \{in\}) \cup \{out\}] \\
&\quad \land\ currentSubTerm' = [currentSubTerm \text{ EXCEPT } ![i] = @ + 1] \\
&\quad \land\ \text{UNCHANGED } \langle messages, serverVars, candidateVars, nextIndex, \\
&\qquad\qquad matchIndex, witnessSubterm, log, commitIndex, \\
&\qquad\qquad witnessVars\rangle
\end{aligned}
$$

Figure 2.5: Replicaiton set adjustment

possible. To achieve this, leader can track the status of each peer regular server by monitoring their responses. If leader receives no response from a regular peer server over an *electiontimeout*, it assumes the regular server is unreachable and initiates a replication set adjustment. Let *Reachable* and *Unreachable* represent the set of reachable regular servers and unreachable regular servers from the leader's perspective. A practical replication set adjustment can be implemented as below:

1. When a regular server becomes a leader, its replication set is initialized to all regular servers *RegularServer* in the leader's configuration.

2. If all servers in *RegularServer* are reachable from the leader, leader changes its replication set to *RegularServer*, i.e.,

$$\forall s \in \mathit{RegularServer} : s \in \mathit{Reachable} \implies \mathit{ReplicationSet}' = \mathit{RegularServer}$$

3. Leader swaps one unreachable regular server inside its replication set with the reachable regular server or witness outside, i.e.,

$$\begin{aligned} \exists x \in \mathit{ReplicationSet}, y \in \mathit{Server} \setminus \mathit{ReplicationSet} : \ & \land\, x \in \mathit{Unreachable} \\ & \land\, \lor\, y \in \mathit{Reachable} \\ & \quad\ \lor\, y = \mathit{witness} \\ \implies \mathit{ReplicationSet}' = \mathit{Server} \setminus \{x\} & \end{aligned}$$

4. Leader increments its *currentSubterm* and appends a new empty log entry if its replication set changes.

2.5 Leader Election

When a regular server transitions into candidate, it requests votes from all peer servers including witness. The extended Raft algorithm adheres to the exact same rules as those in Raft when it comes to requesting and granting votes between a candidate and a regular server. However, the candidate does not request a vote from the witness until it has received subquorum votes from regular servers. Figure 2.6 describes the new *RequestWitnessVote* action for a candidate to request vote from a witness.

Considering that the witness does not persist the full log prefix, the extended Raft algorithm employs a new set of rules for the witness in leader elections.

Candidate i sends witness a RequestWitnessVote request.

$$
\begin{array}{l}
RequestWitnessVote(i) \triangleq \\
\quad \land\ state[i] = Candidate \\
\quad \land\ WitnessID \notin votesResponded[i] \\
\quad \land\ (votesGranted[i] \cup \{WitnessID\}) \in Quorum \\
\quad \land\ Send([\begin{array}[t]{@{}ll}
mtype & \mapsto RequestWitnessVoteRequest, \\
mterm & \mapsto currentTerm[i], \\
mlastLogTerm & \mapsto LastTerm(log[i]), \\
mlastLogSubterm & \mapsto LastSubterm(log[i]), \\
mvotesGranted & \mapsto votesGranted[i], \\
msource & \mapsto i, \\
mdest & \mapsto WitnessID])
\end{array} \\
\quad \land\ \text{UNCHANGED } \langle serverVars,\ candidateVars,\ leaderVars,\ logVars, \\
\qquad\qquad\qquad\quad witnessVars\rangle
\end{array}
$$

Figure 2.6: RequestWitnessVote

Figure 2.7 describes the new action for the witness to handle a vote request from a candidate. The *RequestWitnessVoteRequest* message includes the *term* and *subterm* of the last log entry in the candidate, as well as the votes the candidate has already received in the current election. The witness votes for a candidate if it has not casted vote in the current term and any of the following conditions is true:

- Candidate's last log entry has larger term, i.e. $m.mlastLogTerm > witnessLastLogTerm$.
- Candidate's last log entry has same term but larger subterm, i.e. $m.mlastLogTerm = witnessLastLogTerm \land m.lastLogSubterm > witnessLastLogSubterm$.
- Candidate's last log entry has same term and subterm comparing to witness, and all subqorum votes it got are from regular servers in witness's replication set, i.e.

$$m.mlastLogTerm = witnessLastLogTerm$$
$$m.mlastLogSubterm = witnessLastLogSubterm$$
$$m.mvotesGranted \subseteq witnessReplicationSet$$

Witness receives a RequestWitnessVote request from server j with m.mterm ¡= currentTerm[WitnessID].

$$
\begin{array}{l}
\mathit{HandleRequestWitnessVoteRequest}(j,\ m) \triangleq \\
\quad \textsc{let}\ \mathit{logOk} \triangleq \lor\ m.\mathit{mlastLogTerm} > \mathit{witnessLastLogTerm} \\
\qquad\qquad\qquad\quad \lor\ \land\ m.\mathit{mlastLogTerm} = \mathit{witnessLastLogTerm} \\
\qquad\qquad\qquad\qquad\ \ \land\ m.\mathit{mlastLogSubterm} > \mathit{witnessLastLogSubterm} \\
\qquad\qquad\qquad\quad \lor\ \land\ m.\mathit{mlastLogTerm} = \mathit{witnessLastLogTerm} \\
\qquad\qquad\qquad\qquad\ \ \land\ m.\mathit{mlastLogSubterm} = \mathit{witnessLastLogSubterm} \\
\qquad\qquad\qquad\qquad\ \ \land\ m.\mathit{mvotesGranted} \subseteq \mathit{witnessReplicationSet} \\
\qquad\quad\ \ \mathit{grant} \triangleq \land\ m.\mathit{mterm} = \mathit{currentTerm}[\mathit{WitnessID}] \\
\qquad\qquad\qquad\quad \land\ \mathit{logOk} \\
\qquad\qquad\qquad\quad \land\ \mathit{votedFor}[\mathit{WitnessID}] \in \{\mathit{Nil},\ j\} \\
\quad \textsc{in}\ \ \land\ m.\mathit{mterm} \leq \mathit{currentTerm}[\mathit{WitnessID}] \\
\qquad\quad \land\ \lor\ \mathit{grant}\ \ \land\ \mathit{votedFor}' = [\mathit{votedFor}\ \textsc{except}\ ![\mathit{WitnessID}] = j] \\
\qquad\qquad \lor\ \neg\mathit{grant} \land \textsc{unchanged}\ \mathit{votedFor} \\
\qquad\quad \land\ \mathit{Reply}([\mathit{mtype} \mapsto \mathit{RequestVoteResponse}, \\
\qquad\qquad\qquad\quad \mathit{mterm} \mapsto \mathit{currentTerm}[\mathit{WitnessID}], \\
\qquad\qquad\qquad\quad \mathit{mvoteGranted} \mapsto \mathit{grant},
\end{array}
$$

mlog is used just for the ‘elections’ history variable for the proof. It would not exist in a real implementation.

$$
\begin{array}{l}
\qquad\qquad\qquad\quad \mathit{mlog} \mapsto \mathit{log}[\mathit{WitnessID}], \\
\qquad\qquad\qquad\quad \mathit{msource} \mapsto \mathit{WitnessID}, \\
\qquad\qquad\qquad\quad \mathit{mdest} \mapsto j], \\
\qquad\qquad\qquad\quad m) \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{state},\ \mathit{currentTerm},\ \mathit{candidateVars},\ \mathit{leaderVars}, \\
\qquad\qquad\qquad\qquad\quad \mathit{logVars},\ \mathit{witnessVars} \rangle
\end{array}
$$

Figure 2.7: HandleRequestVoteToWitness

2.6 Membership Reconfiguration

Cluster configuration change can be done in both simple and complex approach in Raft algorithm. The simple approach changes one server at a time. And the complex approach allows arbitrary configuration changes at one time.

The key to the safety of configuration change in Raft is preserving overlap between any majority of the old cluster and any majority of the new cluster. This overlap prevents the cluster from splitting into two independent majorities where two leaders being elected in same term, since otherwise the overlapped server would be able to vote two servers in a term, which is a a contradiction of the specification. This applies to the extended Raft algorithm too because a new leader also needs quorum voters to succeed in an election.Therefore, it is safe to apply same membership configuration methods in Raft algorithm to the extended Raft algorithm either with a sequence of single-server membership change or with an intermediate joint configuration that overlaps both old and new membership configurations.

3 Witness Implementation

With the extended Raft algorithm, the witness possesses the following properties that make it particularly suitable for implementation as a storage object:

- Witness does not have volatile variables, implying that all states within the witness are persisted in durable storage.
- Witness only persists a small amount of metadata. Neither user data nor the full log prefix metadata are stored in witness. This results in a trivial and predictable capacity requirement.
- Witness is rarely accessed. The shortcut replication ensures that leader only needs to write to witness when its replication set changes. Therefore, witness may only need to be accessed when faults change (e.g. a server shuts down or recovers) or a candidate requests its vote. In a relatively stable cluster, neither of these events occur frequently.

The absence of volatile variables allows the synchronization of witness states to be distributed across all regular servers. For each log replication between leader and witness, the *HandleAppendEntriesToWitnessRequest* action can be implemented in the leader as a single operation that loads states from the witness's durable storage, make new states according to *HandleAppendEntriesToWitnessRequest*, and writes back new states to the witness's durable storage. Similarly for voting between candidate and witness, the *HandleRequestWitnessVoteRequest* action can be implemented in the candidate as a single operation that loads states from the witness's durable storage, make new states according to *HandleRequestVoteToWitness*, and writes back the new states to the witness's durable storage. These two implementations can be synchronized using an optimistic concurrency control scheme since they both occur infrequently.

Figure 3.1 illustrates an implementation using a share (NFS or SMB) as a witness. We associate a version number with the states, starting from 0. Each modification to the states increases the version by 1. We can then persist each version of the witness states in a file uniquely named after the version number (e.g. <version>.st). Whenever *HandleAppendEntriesToWitnessRequest* or *HandleRequestVoteToWitness* are invoked, a regular server (leader or candidate) loads the witness states from the file with the largest version number, calculates the new states, and saves the new states back to a distinct temporary file (e.g. <server id>.<version>.st). Then, the regular server tries to create a hard link from the temporary file to the desired versioned state file (i.e. <version+1>.st). As long as the file system guarantees atomicity in creating hardlinked files, the regular server will only succeed if the target versioned state file does not exist. If any other server has finished update states in the same version, the process needs to be repeated from the beginning. The optimistic concurrency control eliminates the need for locking overhead, which is challenging to implement in practical applications. The infrequency of witness visits makes such a scheme very suitable.

In addition to share, the witness functionality can also be incorporated into cloud storage, such as Azure Storage or AWS S3. Many cloud storage vendors provide optimistic concurrency mechanism for updating object data. For instance, Azure Blob Storage allows clients to update a blob object using the original ETag, combined with a conditional header, to ensure that updates only occur if the ETag remains unchanged. This guarantees that no other client has updated the blob object concurrently. The infrequent access requirements and trivial payload size for witness make cloud storage an ideal solution for this function. Consequently, it provides a broader range of implementation options for end-users.

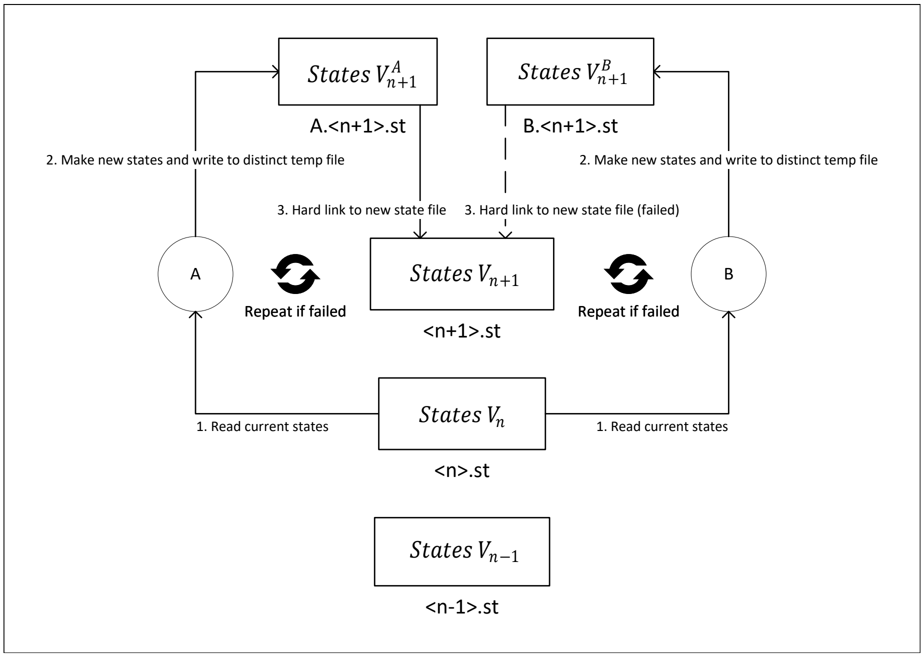

Figure 3.1: Implement witness as a share

4 Discussion

4.1 Compatibility

In a cluster without witness, the extended Raft algorithm operates identically to the Raft algorithm. A cluste can be formed with mixed servers running with either Raft or the extended Raft algorithms. This compatibility facilitates a seamless upgrade from a cluster based on the Raft algorithm to one using the extended Raft algorithm in practical applications. A binary upgrade can be performed in the existing cluster to incorporate the extended Raft algorithm. Following this, the upgraded cluster functions similarly to a typical Raft-based cluster, without the inclusion of a witness. Once all servers have been upgraded to support the extended Raft algorithm, a membership reconfiguration operation can be performed to integrate a witness into the cluster. Conversely, a cluster functioning with the extended Raft algorithm can be reverted back to Raft algorithm through a binary downgrade, which is possible after removal of witness from the cluster.

The compatibility is crucial for real-world applications. It eliminates the necessity to establish a new cluster and migrate data from the existing cluster to take advantage of the benefits offered by a witness. As a result, this minimizes disruptions and enhances the operational efficiency of these applications.

4.2 Availability

The Raft algorithm ensures the presence of a leader that preserves all committed log entries and continues to commit incoming requests, provided that the cluster has a quorum of connected, healthy servers. This implies that if a follower fails, the leader cannot commit if the remaining followers cannot form a quorum with the leader. Additionally, if a leader fails, a new leader can be elected among the other servers that form a quorum.

However, this is a slightly different in the extended Raft algorithm.

4.3 Cascading Failures

The extended Raft algorithm's correctness doesn't guarantee that a leader can always be elected. This predicament often arises when faults occur, leading to a situation where no regular server has a more up-to-date log than witness. In other words, witness would have been the only feasible next leader if it weren't for the lack of a full log prefix. This typically happens as a result of cascading failures where servers go down sequentially.

For instance, consider a cluster composed of two servers (A and B) and a witness. If A is the leader and a sequence of events unfold as described below, the cluster will find itself in a situation where A is the only functioning regular server, but its log doesn't contain the new committed log entry (made by B). The state in the witness prevents it from casting a vote for A, leaving the cluster in a situation without a leader. However, this doesn't violate the correctness of the extended Raft algorithm. And a new leader can always be elected once B restarts.

1. A is shut down
2. B becomes leader
3. B commits a new entry (hence writes states to witness)
4. A starts up
5. B is shut down before it replicating the new committed entry to A

In real-world scenarios, when a server shuts down, its subsystems do not cease functioning in the same time. For instance, NIC connecting to the witness (NIC_w) may stop after the other NIC that connects to regular servers (NIC_s). Moreover, the server's OS ceases operation after all NICs go down. Considering the same cluster as before, if A was operating as leader in the extended Raft algorithm, unfunctional NIC_s would prompt it to adjust its replication set and advance the subterm. A new log entry may then be committed in the new subterm, with its term and subterm recorded to the witness through the still-functioning NIC_w. Now after A fully shuts down, B won't be elected as the new leader, since the witness, containing more up-to-date states, rejects casting a vote for it.

This particular issue arises due to the limited role of the witness. However, it's a necessary trade-off to minimize the overall footprint and maintain reasonable performance when faults occur. It's important to note that this issue arises in very rare cases. Adding an extra delay in *HandleAppendEntriesToWitnessRequest* can greately reduce the chance of the occurence. The impact to performance is trivial, as most commits do not involve writing to the witness.

4.4 Catching up upon replication set adjustment

When a follower server's log significantly lags behind the leader's, it may take a considerable amount of time for the follower server to catch up. Until then, new log entries in the leader will not be acknowledged by that follower server. If the live servers merely satisfy the minimal quorum number, the leader won't be able to commit until all servers catch up. This situation arises in Raft when a new server is added to a cluster, and the solution is to add the new server as a learner that receives the log but is not a part of the current membership configuration.

This situation can also occur in the extended Raft algorithm when subterm changes. When a server outside replication set recovers from fault state, the leader may rejoin it to the replication set. If there is only a quorum of live servers (including the witness) in the current replication set, the leader won't be able to commit new log entries until all servers catch up. For instance, in the cluster above, if A commits many entries during B's downtime, B's log will significantly lag behind A. After B recovers, A will advance its subterm and change its replication set to $\{A, B\}$. Now, any new entry in A will not be committed until it receives 2 acknowledgements, i.e., both A and B must acknowledge. As a result, the cluster won't be able to commit any entry because it will take B a long time to catch up with A and acknowledge.

The solution to this issue is similar to the learner approach: leader does not change its replication set to add a regular server until that server's log catches up with the leader.

5 Specification

```
------------------------------ MODULE extendedraft ------------------------------
 This is the formal specification for the extended Raft consensus algorithm.
 < TODO: Copyright Info >

 Copyright 2014 Diego Ongaro.
 This work is licensed under the Creative Commons Attribution - 4.0
 International License https://creativecommons.org/licenses/by/4.0/

EXTENDS Naturals, FiniteSets, Sequences, TLC

 The set of server IDs
CONSTANTS Server

 The set of requests that can go into the log
CONSTANTS Value

 ID of the witness
CONSTANTS WitnessID

 Server states.
CONSTANTS Follower, Candidate, Leader

 A reserved value.
CONSTANTS Nil

 Message types:
CONSTANTS RequestVoteRequest, RequestVoteResponse,
          AppendEntriesRequest, AppendEntriesResponse,
          RequestWitnessVoteRequest, AppendEntriesToWitnessRequest

----------------------------------------------------------------------------------
 Global variables
```

A bag of records representing requests and responses sent from one server to another. *TLAPS* doesn't support the Bags module, so this is a function mapping Message to *Nat*.

VARIABLE *messages*

A history variable used in the proof. This would not be present in an implementation.
Keeps track of successful elections, including the initial logs of the leader and voters' logs. Set of functions containing various things about successful elections (see *BecomeLeader*).

VARIABLE *elections*

A history variable used in the proof. This would not be present in an implementation.
Keeps track of every *log* ever in the system (set of logs).

VARIABLE *allLogs*

The following variables are all per server (functions with domain *Server*).

The server's term number.

VARIABLE *currentTerm*

The server's state (Follower, *Candidate*, or *Leader*).

VARIABLE *state*

The candidate the server voted for in its current term, or Nil if it hasn't voted for any.

VARIABLE *votedFor*

$serverVars \triangleq \langle currentTerm, state, votedFor \rangle$

A Sequence of *log* entries. The index into this sequence is the index of the *log* entry. Unfortunately, the Sequence module defines $Head(s)$ as the entry with index 1, so be careful not to use that!

VARIABLE *log*

The index of the latest entry in the *log* the state machine may apply.

VARIABLE *commitIndex*

$logVars \triangleq \langle log, commitIndex \rangle$

The following variables are used only on candidates:
The set of servers from which the candidate has received a *RequestVote*

response in its *currentTerm*.

VARIABLE *votesResponded*

The set of servers from which the candidate has received a vote in its *currentTerm*.

VARIABLE *votesGranted*

A history variable used in the proof. This would not be present in an implementation.

Function from each server that voted for this candidate in its *currentTerm* to that voter's *log*.

VARIABLE *voterLog*

$candidateVars \triangleq \langle votesResponded, votesGranted, voterLog \rangle$

The following variables are used only on leaders:

The next entry to send to each follower.

VARIABLE *nextIndex*

The latest entry that each follower has acknowledged is the same as the leader's. This is used to calculate *commitIndex* on the leader.

VARIABLE *matchIndex*

Leader's replication set

VARIABLE *replicationSet*

Leader's *subterm* number

VARIABLE *currentSubterm*

The latest *subterm* that this leader has written to the witness

VARIABLE *witnessSubterm*

$leaderVars \triangleq \langle nextIndex, matchIndex, elections,$
$\quad replicationSet, currentSubterm, witnessSubterm \rangle$

The following variables are persisted only on witness:

The latest replication set sent from leader

VARIABLE *witnessReplicationSet*

The latest term of replicated *log* entry

VARIABLE *witnessLastLogTerm*

The latest *subterm* of replicated *log* entry

VARIABLE *witnessLastLogSubterm*

$witnessVars \triangleq \langle witnessReplicationSet,$
$\quad witnessLastLogTerm, witnessLastLogSubterm \rangle$

End of per server variables.

All variables; used for stuttering (asserting state hasn't changed).

$$vars \triangleq \langle messages,\ allLogs,\ serverVars,\ candidateVars,\ leaderVars,\ logVars,\ witnessVars \rangle$$

Helpers

The set of all quorums. This just calculates simple majorities, but the only important property is that every quorum overlaps with every other.

$$Quorum \triangleq \{i \in \text{SUBSET}\ (Server) : Cardinality(i) * 2 > Cardinality(Server)\}$$

The term of the last entry in a *log*, or 0 if the *log* is empty.

$$LastTerm(xlog) \triangleq \text{IF}\ Len(xlog) = 0\ \text{THEN}\ 0\ \text{ELSE}\ xlog[Len(xlog)].term$$

The *subterm* of the last entry in a *log*, or 0 if the *log* is empty.

$$LastSubTerm(xlog) \triangleq \text{IF}\ Len(xlog) = 0\ \text{THEN}\ 0\ \text{ELSE}\ xlog[Len(xlog)].subterm$$

Helper for *Send* and *Reply*. Given a message *m* and bag of messages, return a new bag of messages with one more *m* in it.

$$\begin{aligned} &WithMessage(m,\ msgs) \triangleq \\ &\quad \text{IF}\ m \in \text{DOMAIN}\ msgs\ \text{THEN} \\ &\qquad [msgs\ \text{EXCEPT}\ ![m] = msgs[m] + 1] \\ &\quad \text{ELSE} \\ &\qquad msgs\ @@\ (m :> 1) \end{aligned}$$

Helper for *Discard* and *Reply*. Given a message *m* and bag of messages, return a new bag of messages with one less *m* in it.

$$\begin{aligned} &WithoutMessage(m,\ msgs) \triangleq \\ &\quad \text{IF}\ m \in \text{DOMAIN}\ msgs\ \text{THEN} \\ &\qquad [msgs\ \text{EXCEPT}\ ![m] = msgs[m] - 1] \\ &\quad \text{ELSE} \\ &\qquad msgs \end{aligned}$$

Add a message to the bag of messages.

$$Send(m) \triangleq messages' = WithMessage(m,\ messages)$$

Remove a message from the bag of messages. Used when a server is done processing a message.

$$Discard(m) \triangleq messages' = WithoutMessage(m,\ messages)$$

```
Combination of Send and Discard
Reply(response, request) ≜
    messages' = WithoutMessage(request, WithMessage(response, messages))

Return the minimum value from a set, or undefined if the set is empty.
Min(s) ≜ CHOOSE x ∈ s : ∀ y ∈ s : x ≤ y
Return the maximum value from a set, or undefined if the set is empty.
Max(s) ≜ CHOOSE x ∈ s : ∀ y ∈ s : x ≥ y

----------------------------------------------------------------------

Define initial values for all variables

InitHistoryVars ≜ ∧ elections = {}
                  ∧ allLogs   = {}
                  ∧ voterLog = [i ∈ Server ↦ [j ∈ {} ↦ ⟨⟩]]
InitServerVars ≜ ∧ currentTerm = [i ∈ Server ↦ 1]
                 ∧ state       = [i ∈ Server ↦ Follower]
                 ∧ votedFor    = [i ∈ Server ↦ Nil]
InitCandidateVars ≜ ∧ votesResponded = [i ∈ Server ↦ {}]
                    ∧ votesGranted   = [i ∈ Server ↦ {}]
The values nextIndex[i][i] and matchIndex[i][i] are never read, since the
leader does not send itself messages. It's still easier to include these
in the functions.
InitLeaderVars ≜ ∧ nextIndex  = [i ∈ Server ↦ [j ∈ Server ↦ 1]]
                 ∧ matchIndex = [i ∈ Server ↦ [j ∈ Server ↦ 0]]
                 ∧ replicationSet = Server \ {WitnessID}
                 ∧ currentSubterm = 0
InitLogVars ≜ ∧ log         = [i ∈ Server ↦ ⟨⟩]
              ∧ commitIndex = [i ∈ Server ↦ 0]
InitWitnessVars ≜ ∧ witnessReplicationSet = {}
                  ∧ witnessLastLogTerm = 0
                  ∧ witnessLastLogSubterm = 0
Init ≜ ∧ messages = [m ∈ {} ↦ 0]
       ∧ InitHistoryVars
       ∧ InitServerVars
       ∧ InitCandidateVars
       ∧ InitLeaderVars
       ∧ InitLogVars
```

```
    ∧ InitWitnessVars
────────────────────────────────────────────────────────────────
Define state transitions

Server i restarts from stable storage.
It loses everything but its currentTerm, votedFor, and log.
Restart(i) ≜
    ∧ i                ≠ WitnessID
    ∧ state'           = [state EXCEPT ![i] = Follower]
    ∧ votesResponded'  = [votesResponded EXCEPT ![i] = {}]
    ∧ votesGranted'    = [votesGranted EXCEPT ![i] = {}]
    ∧ voterLog'        = [voterLog EXCEPT ![i] = [j ∈ {} ↦ ⟨⟩]]
    ∧ nextIndex'       = [nextIndex EXCEPT ![i] = [j ∈ Server ↦ 1]]
    ∧ matchIndex'      = [matchIndex EXCEPT ![i] = [j ∈ Server ↦ 0]]
    ∧ commitIndex'     = [commitIndex EXCEPT ![i] = 0]
    ∧ currentSubterm'  = [currentSubterm EXCEPT ![i] = 0]
    ∧ replicationSet'  = [replicationSet EXCEPT ![i]   = Server]
    ∧ witnessSubterm'  = [witnessSubterm EXCEPT ![i] = 0]
    ∧ UNCHANGED ⟨messages, currentTerm, votedFor, log, elections, witnessVars⟩

Server i times out and starts a new election.
Timeout(i) ≜ ∧ i ≠ WitnessID
             ∧ state[i] ∈ {Follower, Candidate}
             ∧ state' = [state EXCEPT ![i] = Candidate]
             ∧ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
              Most implementations would probably just set the local vote
              atomically, but messaging localhost for it is weaker.
             ∧ votedFor' = [votedFor EXCEPT ![i] = Nil]
             ∧ votesResponded' = [votesResponded EXCEPT ![i] = {}]
             ∧ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
             ∧ voterLog'       = [voterLog EXCEPT ![i] = [j ∈ {} ↦ ⟨⟩]]
             ∧ UNCHANGED ⟨messages, leaderVars, logVars, witnessVars⟩

Candidate i sends j a RequestVote request.
RequestVote(i, j) ≜
    ∧ state[i] = Candidate
    ∧ j ≠ WitnessID
```

```
        ∧ j ∉ votesResponded[i]
        ∧ Send([mtype          ↦ RequestVoteRequest,
                mterm          ↦ currentTerm[i],
                mlastLogTerm   ↦ LastTerm(log[i]),
                mlastLogIndex  ↦ Len(log[i]),
                msource        ↦ i,
                mdest          ↦ j])
        ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars, witnessVars⟩

  Candidate i sends witness a RequestWitnessVote request.
RequestWitnessVote(i) ≜
        ∧ state[i] = Candidate
        ∧ WitnessID ∉ votesResponded[i]
        ∧ (votesGranted[i] ∪ {WitnessID}) ∈ Quorum
        ∧ Send([mtype              ↦ RequestWitnessVoteRequest,
                mterm              ↦ currentTerm[i],
                mlastLogTerm       ↦ LastTerm(log[i]),
                mlastLogSubterm    ↦ LastSubterm(log[i]),
                mvotesGranted      ↦ votesGranted[i],
                msource            ↦ i,
                mdest              ↦ WitnessID])
        ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars,
                     witnessVars⟩

  Leader i sends j an AppendEntries request containing up to 1 entry.
  While implementations may want to send more than 1 at a time, this spec uses
  just 1 because it minimizes atomic regions without loss of generality.
AppendEntries(i, j) ≜
        ∧ i ≠ j
        ∧ state[i] = Leader
        ∧ LET prevLogIndex ≜ nextIndex[i][j] − 1
              prevLogTerm ≜ IF prevLogIndex > 0 THEN
                                 log[i][prevLogIndex].term
                             ELSE
                                 0
               Send up to 1 entry, constrained by the end of the log.
              lastEntry ≜ Min({Len(log[i]), nextIndex[i][j]})
```

```
                entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)
         IN  Send([mtype          ↦ AppendEntriesRequest,
                   mterm          ↦ currentTerm[i],
                   mprevLogIndex  ↦ prevLogIndex,
                   mprevLogTerm   ↦ prevLogTerm,
                   mentries       ↦ entries,
                    mlog is used as a history variable for the proof.
                    It would not exist in a real implementation.
                   mlog           ↦ log[i],
                   mcommitIndex   ↦ Min({commitIndex[i], lastEntry}),
                   msource        ↦ i,
                   mdest          ↦ j])
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars,
                   witnessVars⟩

 Leader i sends witness an AppendEntriesToWitness request with log entry
 in current subterm acknowledged by a subquorum in current replication set.
AppendEntriesToWitness(i) ≜
    ∧ state[i] = Leader
    ∧ WitnessID ∈ replicationSet[i]
    ∧ LET Agree(index) ≜
                {i} ∪ {k ∈ replicationSet[i] : matchIndex[i][k] ≥ index}
          IsAgreed(k) ≜
                ∧ log[i][k].term    = currentTerm[i]
                ∧ log[i][k].subterm = currentSubterm[i]
                ∧ ({WitnessID} ∪ Agree(k)) ∈ Quorum
          agreeIndexes ≜ {k ∈ 1 .. Len(log[i]) : IsAgreed(k)}
          lastEntry    ≜ Max(agreedIndex)
       IN ∧ agreeIndexes ≠ {}
          ∧ ∨  witnessSubterm never exceeds currentSubterm following specification
               ∧ currentSubterm[i] > witnessSubterm
               ∧ Send([mtype           ↦ AppendEntriesToWitnessRequest,
                       mterm           ↦ currentTerm[i],
                       mlogTerm        ↦ log[i][lastEntry].term,
                       mlogSubterm     ↦ log[i][lastEntry].subterm,
                       mreplicationSet ↦ replicaitonSet[i],
                       mindex          ↦ lastEntry,
```

```
                              mlog and mentries are used as history variable for
                              the proof. They do not exist in a real implementation.
                             mlog           ↦ log[i],
                             mentries       ↦ SubSeq(log[i], 1, lastEntry),
                             msource        ↦ i,
                             mdest          ↦ WitnessID])
                   ∧ UNCHANGED ⟨leaderVars⟩
                ∨  shortcut replication
                   ∧ currentSubterm[i] = witnessSubterm[i]
                   ∧ Send([mtype            ↦ AppendEntriesResponse,
                             mterm          ↦ currentTerm[i],
                             msuccess       ↦ TRUE,
                             mmatchIndex    ↦ lastEntry,
                             msource        ↦ WitnessID,
                             mdest          ↦ i])
    ∧ UNCHANGED ⟨serverVars, candidateVars, logVars, witnessVars⟩

 Candidate i transitions to leader.
BecomeLeader(i) ≜
    ∧ state[i] = Candidate
    ∧ votesGranted[i] ∈ Quorum
    ∧ state'        = [state EXCEPT ![i] = Leader]
    ∧ nextIndex'    = [nextIndex EXCEPT ![i] =
                         [j ∈ Server ↦ Len(log[i]) + 1]]
    ∧ matchIndex'   = [matchIndex EXCEPT ![i] =
                         [j ∈ Server ↦ 0]]
    ∧ elections'    = elections ∪
                         {[eterm      ↦ currentTerm[i],
                           eleader    ↦ i,
                           elog       ↦ log[i],
                           evotes     ↦ votesGranted[i],
                           evoterLog  ↦ voterLog[i]]}
    ∧ UNCHANGED ⟨messages, currentTerm, votedFor, candidateVars, logVars, witnessVars⟩

 Leader i receives a client request to add v to the log.
ClientRequest(i, v) ≜
    ∧ state[i] = Leader
```

```
    ∧ LET entry    ≜ [term       ↦ currentTerm[i],
                      subterm    ↦ currentSubterm[i],
                      value      ↦ v]
          newLog ≜ Append(log[i], entry)
      IN  log′ = [log EXCEPT ![i] = newLog]
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars,
                 leaderVars, commitIndex, witnessVars⟩
```

Leader i advances its *commitIndex*.
This is done as a separate step from handling *AppendEntries* responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.

```
AdvanceCommitIndex(i) ≜
    ∧ state[i] = Leader
    ∧ LET  The set of servers that agree up through index.
          Agree(index) ≜ {i} ∪ {k ∈ Server :
                                   matchIndex[i][k] ≥ index}
           The maximum indexes for which a quorum agrees
          agreeIndexes ≜ {index ∈ 1 .. Len(log[i]) :
                              Agree(index) ∈ Quorum}
           New value for commitIndex′[i]
          newCommitIndex ≜
             IF ∧ agreeIndexes ≠ {}
                ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
              THEN
                Max(agreeIndexes)
              ELSE
                commitIndex[i]
      IN  commitIndex′ = [commitIndex EXCEPT ![i] = newCommitIndex]
    ∧ UNCHANGED ⟨messages, serverVars, candidateVars, leaderVars, log, witnessVars⟩
```

Adjust replication set on leader i. This action updates replication set
by swapping items inside and outside. While implementations may change
replication set in various ways, the spec uses this simple swapping to
minimize atomic regions without loss of generality.

```
AdjustReplicationSet(i) ≜
    ∧ state[i] = Leader
```

```
      ∧ replicationSet[i] ≠ Server
      Swap server outside replication set with some server inside
      ∧ LET in  ≜ CHOOSE x ∈ replicationSet[i] : x ≠ i
            out ≜ CHOOSE x ∈ Server \ replicationSet[i] : TRUE
        IN  replicationSet' =
              [replicationSet EXCEPT ![i] = (@ \ {in}) ∪ {out}]
      ∧ currentSubTerm' = [currentSubTerm EXCEPT ![i] = @ + 1]
      ∧ UNCHANGED ⟨messages, serverVars, candidateVars, nextIndex,
                   matchIndex, witnessSubterm, log, commitIndex,
                   witnessVars⟩
```

```
  Message handlers
  i = recipient, j = sender, m = message

  Server i receives a RequestVote request from server j with
  m.mterm ≤ currentTerm[i].
HandleRequestVoteRequest(i, j, m) ≜
    LET logOk ≜ ∨ m.mlastLogTerm > LastTerm(log[i])
                ∨ ∧ m.mlastLogTerm = LastTerm(log[i])
                  ∧ m.mlastLogIndex ≥ Len(log[i])
        grant ≜ ∧ m.mterm = currentTerm[i]
                ∧ logOk
                ∧ votedFor[i] ∈ {Nil, j}
    IN  ∧ m.mterm ≤ currentTerm[i]
        ∧ ∨ grant  ∧ votedFor' = [votedFor EXCEPT ![i] = j]
          ∨ ¬grant ∧ UNCHANGED votedFor
        ∧ Reply([mtype         ↦ RequestVoteResponse,
                 mterm         ↦ currentTerm[i],
                 mvoteGranted  ↦ grant,
                  mlog is used just for the elections history variable for
                  the proof. It would not exist in a real implementation.
                 mlog          ↦ log[i],
                 msource       ↦ i,
                 mdest         ↦ j],
                 m)
        ∧ UNCHANGED ⟨state, currentTerm, candidateVars, leaderVars, logVars, witnessVars⟩
```

```
Witness receives a RequestWitnessVote request from server j with
m.mterm ≤ currentTerm[WitnessID].
HandleRequestWitnessVoteRequest(j, m) ≜
    LET logOk ≜ ∨ m.mlastLogTerm > witnessLastLogTerm
                ∨ ∧ m.mlastLogTerm = witnessLastLogTerm
                  ∧ m.mlastLogSubterm > witnessLastLogSubterm
                ∨ ∧ m.mlastLogTerm = witnessLastLogTerm
                  ∧ m.mlastLogSubterm = witnessLastLogSubterm
                  ∧ m.mvotesGranted ⊆ witnessReplicationSet
        grant ≜ ∧ m.mterm = currentTerm[WitnessID]
                ∧ logOk
                ∧ votedFor[WitnessID] ∈ {Nil, j}
    IN  ∧ m.mterm ≤ currentTerm[WitnessID]
        ∧ ∨ grant  ∧ votedFor' = [votedFor EXCEPT ![WitnessID] = j]
          ∨ ¬grant ∧ UNCHANGED votedFor
        ∧ Reply([mtype        ↦ RequestVoteResponse,
                 mterm        ↦ currentTerm[WitnessID],
                 mvoteGranted ↦ grant,
                  mlog is used just for the elections history variable for
                  the proof. It would not exist in a real implementation.
                 mlog         ↦ log[WitnessID],
                 msource      ↦ WitnessID,
                 mdest        ↦ j],
                 m)
        ∧ UNCHANGED ⟨state, currentTerm, candidateVars, leaderVars,
                     logVars, witnessVars⟩

Server i receives a RequestVote response from server j with
m.mterm = currentTerm[i].
HandleRequestVoteResponse(i, j, m) ≜
    This tallies votes even when the current state is not Candidate, but
    they won't be looked at, so it doesn't matter.
    ∧ m.mterm = currentTerm[i]
    ∧ votesResponded' = [votesResponded EXCEPT ![i] =
                                  votesResponded[i] ∪ {j}]
    ∧ ∨ ∧ m.mvoteGranted
        ∧ votesGranted' = [votesGranted EXCEPT ![i] =
```

```
                                        votesGranted[i] ∪ {j}]
              ∧ voterLog′ = [voterLog EXCEPT ![i] =
                                     voterLog[i] @@ (j :> m.mlog)]
           ∨ ∧ ¬m.mvoteGranted
              ∧ UNCHANGED ⟨votesGranted, voterLog⟩
        ∧ Discard(m)
        ∧ UNCHANGED ⟨serverVars, votedFor, leaderVars, logVars, witnessVars⟩

   Server i receives an AppendEntries request from server j with
   m.mterm ≤ currentTerm[i]. This just handles m.entries of length 0 or 1, but
   implementations could safely accept more by treating them the same as
   multiple independent requests of 1 entry.
HandleAppendEntriesRequest(i, j, m) ≜
    LET logOk ≜  ∨ m.mprevLogIndex = 0
                 ∨ ∧ m.mprevLogIndex > 0
                   ∧ m.mprevLogIndex ≤ Len(log[i])
                   ∧ m.mprevLogTerm = log[i][m.mprevLogIndex].term
    IN   ∧ m.mterm ≤ currentTerm[i]
         ∧ ∨ ∧  reject request
                 ∨ m.mterm < currentTerm[i]
                 ∨ ∧ m.mterm = currentTerm[i]
                   ∧ state[i] = Follower
                   ∧ ¬logOk
              ∧ Reply([mtype          ↦ AppendEntriesResponse,
                       mterm          ↦ currentTerm[i],
                       msuccess       ↦ FALSE,
                       mmatchIndex    ↦ 0,
                       msource        ↦ i,
                       mdest          ↦ j],
                       m)
              ∧ UNCHANGED ⟨serverVars, logVars⟩
           ∨  return to follower state
              ∧ m.mterm = currentTerm[i]
              ∧ state[i] = Candidate
              ∧ state′ = [state EXCEPT ![i] = Follower]
              ∧ UNCHANGED ⟨currentTerm, votedFor, logVars, messages⟩
           ∨  accept request
```

```
        ∧ m.mterm = currentTerm[i]
        ∧ state[i] = Follower
        ∧ logOk
        ∧ LET index ≜ m.mprevLogIndex + 1
           IN   ∨ already done with request
                  ∧ ∨ m.mentries = ⟨⟩
                    ∨ ∧ m.mentries ≠ ⟨⟩
                      ∧ Len(log[i]) ≥ index
                      ∧ log[i][index].term = m.mentries[1].term
                    This could make our commitIndex decrease (for
                    example if we process an old, duplicated request),
                    but that doesn't really affect anything.
                  ∧ commitIndex′ = [commitIndex EXCEPT ![i] =
                                           m.mcommitIndex]
                  ∧ Reply([mtype          ↦ AppendEntriesResponse,
                           mterm          ↦ currentTerm[i],
                           msuccess       ↦ TRUE,
                           mmatchIndex    ↦ m.mprevLogIndex +
                                              Len(m.mentries),
                           msource        ↦ i,
                           mdest          ↦ j],
                           m)
                  ∧ UNCHANGED ⟨serverVars, log⟩
                ∨ conflict: remove 1 entry
                  ∧ m.mentries ≠ ⟨⟩
                  ∧ Len(log[i]) ≥ index
                  ∧ log[i][index].term    ≠ m.mentries[1].term
                  ∧ LET new ≜ [index2 ∈ 1 .. (Len(log[i]) − 1) ↦
                                       log[i][index2]]
                     IN   log′ = [log EXCEPT ![i] = new]
                  ∧ UNCHANGED ⟨serverVars, commitIndex, messages⟩
                ∨ no conflict: append entry
                  ∧ m.mentries ≠ ⟨⟩
                  ∧ Len(log[i]) = m.mprevLogIndex
                  ∧ log′ = [log EXCEPT ![i] =
                                  Append(log[i], m.mentries[1])]
                  ∧ UNCHANGED ⟨serverVars, commitIndex, messages⟩
```

```
        ∧ UNCHANGED ⟨candidateVars, leaderVars, witnessVars⟩

Witness receives an AppendEntriesToWitnessRequest from server j with
m.mterm ≤ currentTerm[WitnessID].
HandleAppendEntriesToWitnessRequest(j, m) ≜
    ∧ m.mterm ≤ currentTerm[WitnessID]
    ∧ ∨ reject request
         ∧ m.mterm < currentTerm[WitnessID]
         ∧ Reply([mtype         ↦ AppendEntriesResponse,
                  mterm         ↦ currentTerm[WitnessID],
                  msuccess      ↦ FALSE,
                  mmatchIndex   ↦ 0,
                  msource       ↦ WitnessID,
                  mdest         ↦ j],
                  m)
         ∧ UNCHANGED ⟨serverVars, logVars, witnessVars⟩
      ∨ accept request, always no conflict.
         ∧ m.mterm = currentTerm[WitnessID]
         ∧ ∨ m.mlastLogTerm > witnessLastLogTerm[WitnessID]
           ∨ ∧ m.mlastLogTerm = witnessLastLogTerm[WitnessID]
             ∧ m.mlastLogSubterm > witnessLastLogSubterm[WitnessID]
         ∧ witnessReplicationSet' =
             [witnessReplicationSet EXCEPT ![WitnessID] = m.mreplicationSet]
         ∧ witnessLastLogTerm' =
             [witnessLastLogTerm EXCEPT ![WitnessID] = m.mlastLogTerm]
         ∧ witnessLastLogSubterm' =
             [witnessLastLogSubterm EXCEPT ![WitnessID] = m.mlastLogSubterm]
         log will not be modified in real implementation. It is only
         used here for the proof.
         ∧ log' = [log EXCEPT ![WitnessID] = m.mentries]
         ∧ Reply([mtype         ↦ AppendEntriesResponse,
                  mterm         ↦ currentTerm[WitnessID],
                  msuccess      ↦ TRUE,
                  mmatchIndex   ↦ m.mindex,
                  msource       ↦ WitnessID,
                  mdest         ↦ j],
                  m)
```

```
          ∧ UNCHANGED ⟨serverVars, log⟩
    ∧ UNCHANGED ⟨candidateVars, leaderVars⟩

 Server i receives an AppendEntries response from server j with
 m.mterm = currentTerm[i].
HandleAppendEntriesResponse(i, j, m) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ ∨ ∧ m.msuccess  successful
          ∧ nextIndex′  = [nextIndex  EXCEPT ![i][j] = m.mmatchIndex + 1]
          ∧ matchIndex′ = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
      ∨ ∧ ¬m.msuccess  not successful
          ∧ nextIndex′ = [nextIndex EXCEPT ![i][j] =
                               Max({nextIndex[i][j] − 1, 1})]
          ∧ UNCHANGED ⟨matchIndex⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, logVars, elections, witnessVars⟩

 Any RPC with a newer term causes the recipient to advance its term first.
UpdateTerm(i, j, m) ≜
    ∧ m.mterm > currentTerm[i]
    ∧ currentTerm′ = [currentTerm EXCEPT ![i] = m.mterm]
    ∧ state′       = [state       EXCEPT ![i] = Follower]
    ∧ votedFor′    = [votedFor    EXCEPT ![i] = Nil]
       messages is unchanged so m can be processed further.
    ∧ UNCHANGED ⟨messages, candidateVars, leaderVars, logVars⟩

 Responses with stale terms are ignored.
DropStaleResponse(i, j, m) ≜
    ∧ m.mterm < currentTerm[i]
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩

 Receive a message.
Receive(m) ≜
    LET i ≜ m.mdest
        j ≜ m.msource
    IN   Any RPC with a newer term causes the recipient to advance
         its term first. Responses with stale terms are ignored.
```

```
          ∨ UpdateTerm(i, j, m)
          ∨ ∧ m.mtype = RequestVoteRequest
            ∧ HandleRequestVoteRequest(i, j, m)
          ∨ ∧ m.mtype = RequestWitnessVoteRequest
            ∧ HandleRequestWitnessVoteRequest(j, m)
          ∨ ∧ m.mtype = RequestVoteResponse
            ∧ ∨ DropStaleResponse(i, j, m)
              ∨ HandleRequestVoteResponse(i, j, m)
          ∨ ∧ m.mtype = AppendEntriesRequest
            ∧ HandleAppendEntriesRequest(i, j, m)
          ∨ ∧ m.mtype = AppendEntriesToWitnessReques
            ∧ HandleAppendEntriesToWitnessRequest(j, m)
          ∨ ∧ m.mtype = AppendEntriesResponse
            ∧ ∨ DropStaleResponse(i, j, m)
              ∨ HandleAppendEntriesResponse(i, j, m)

 End of message handlers.
────────────────────────────────────────────────────────────
 Network state transitions

 The network duplicates a message
DuplicateMessage(m) ≜
    ∧ Send(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩

 The network drops a message
DropMessage(m) ≜
    ∧ Discard(m)
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩

────────────────────────────────────────────────────────────
 Defines how the variables may transition.
Next ≜  ∧ ∨ ∃ i ∈ Server : Restart(i)
          ∨ ∃ i ∈ Server : Timeout(i)
          ∨ ∃ i, j ∈ Server : RequestVote(i, j)
          ∨ ∃ i ∈ Server : BecomeLeader(i)
          ∨ ∃ i ∈ Server, v ∈ Value : ClientRequest(i, v)
          ∨ ∃ i ∈ Server : AdvanceCommitIndex(i)
```

```
              ∨ ∃ i, j ∈ Server : AppendEntries(i, j)
              ∨ ∃ i   ∈ Server : AdjustReplicationSet(i)
              ∨ ∃ i   ∈ Server : RequestWitnessVote(i)
              ∨ ∃ i   ∈ Server : AppendEntriesToWitness(i)
              ∨ ∃ m ∈ DOMAIN messages : Receive(m)
              ∨ ∃ m ∈ DOMAIN messages : DuplicateMessage(m)
              ∨ ∃ m ∈ DOMAIN messages : DropMessage(m)
               History variable that tracks every log ever:
           ∧ allLogs' = allLogs ∪ {log[i] : i ∈ Server}

 The specification must start with the initial state and transition according
 to Next.
Spec ≜ Init ∧ □[Next]_vars
```

6 Formal proof

In this section, we aim to prove the key properties of the Raft algorithm (namely, Election Safety, Leader Append-Only, Log Matching, Leader Completeness, and State Machine Safety) within the context of the extended Raft algorithm. These properties will be proven to hold true consistently in the extended Raft algorithm. Our proof will build upon the existing proof of the Raft algorithm, and we will reference lemmas from the Raft proof in the format of "Lemma R.n," where 'n' represents the lemma's index in the Raft proof.

Lemma 1. *Following lemmas are true in the extended Raft algorithm.*

1. *Lemma R.1. Each server's currentTerm monotonically increases.*
2. *Lemma R.2. There is at most one leader per term.*
3. *Lemma R.3. A leader's log monotonically grows during its term.*
4. *Lemma R.4. An* $\langle<\rangle index, term >$ *pair identifies a log prefix.*
5. *Lemma R.5. When a follower appends an entry to its log, its log after the append is a prefix of the leader's log at the time the leader sent the AppendEntries request.*

6. *Lemma R.6. A server's current term is always at least as large as the terms in its log.*
7. *Lemma R.7. The terms of entries grow monotonically in each log.*

Proof. 1. Witness, similar to other regular servers, monotonically increases its term upon receiving a message with a larger *mterm* (action *UpdateTerm*). This immediately validates Lemma R.1 as per the specification.

2. In the extended Raft algorithm, leader is elected by a quorum. Neither regular servers nor the witness vote for different servers within the same term, thus maintaining the truth of Lemma R.2 according to its proof.

3. Leader in the extended Raft algorithm manages its log identically to the Raft algorithm, validating Lemma R.3 since
   (a) Actions *BecomeLeader*, *ClientRequest*, and *AppendEntries* behave same as those in the Raft algorithm. Although the log entry is associated with the current subterm, this is trivial to the proof.
   (b) *AppendEntriesToWitness* is an implementation of the *AppendEntries* action, with batching log prefix to a log entry (that is associated with the current subterm and has received subquorum acknowledgments from the current replication set).

4. Lemma R.4 and Lemma R.5 are valid in the extended Raft algorithm given that:
   (a) Regular servers replicate logs identically to the Raft algorithm. The existence of a subterm in a log entry is trivial to the proof.
   (b) Leader always sends its log's prefix to the witness.
   (c) Witness replaces its log with the log prefix from the leader, hence $log'[WitnessID]$ remains a prefix of $m.mlog$ for m sent to the witness.

5. The validation of Lemma R.6 in the Raft algorithm relies on the servers' behavior in log replication. Therefore, Lemma R.6 holds true in the extended Raft algorithm as:
   (a) Regular servers replicate logs in an identical manner.
   (b) Witness accepts the request with the necessary condition $currentTerm[WitnessID] = m.mterm$.

(c) Witness always replaces its log with the log prefix sent from the leader.

6. The proof of Lemma R.7 depends on Lemma R.5 and Lemma R.6. Thus, Lemma R.7 is also valid in the extended Raft algorithm, as both Lemma R.5 and Lemma R.6 are proven true.

□

Now we have Election Safety property (Lemma R.2), Leader Append-Only property (Lemma R.3), and Log Matching property (Lemma R.4) hold in the extended Raft algorithm.
Similarly, we have following lemmas specific for the extended Raft algorithm.

Lemma 2. *Each leader's currentSubterm monotonically increases during each term:*

$$\begin{aligned}
&\forall i \in \mathit{Server} : \\
&\quad \mathit{currentTerm}[i] = \mathit{currentTerm}'[i] \\
&\quad\quad \implies \mathit{currentSubterm}[i] \leq \mathit{currentSubterm}'[i]
\end{aligned}$$

Proof. This follows immediately from the specification. □

Lemma 3. *The subterms of entries grow monotonically during its termin each log*

$$\begin{aligned}
&\forall l \in \mathit{allLogs} : \\
&\quad \forall \mathit{index} \in 1..(\mathit{Len}(l) - 1) : \\
&\quad\quad l[\mathit{index}].\mathit{term} = l[\mathit{index} + 1].\mathit{term} \\
&\quad\quad\quad \implies l[\mathit{index}].\mathit{subterm} \leq l[\mathit{index} + 1].\mathit{subterm}
\end{aligned}$$

Proof. 1. Initial state: all logs are empty, so the invariant holds.

2. Inductive step: logs change in one of the following ways:

   (a) Case: a leader adds one entry (client request)

      i. The new entry's subterm is $\mathit{currentSubterm}(\mathit{leader})$.

      ii. $\mathit{currentSubterm}$ monotonically increases during $\mathit{currentTerm}$.

      iii. Thus, the new entry's subterm is at least as larger as the subterms during $\mathit{currentTerm}$ in this log, since leader's log monotonically grows during its term.

(b) Case: a follower removes one entry (AppendEntries request)

   i. The invariant still holds, since only the length of the log decreased.

(c) Case: a follower adds one entry (AppendEntries request), or witness replaces its log by log prefix from leader

   i. $log'[follower]$ is a prefix of $m.mlog$ (by Lemma R.5)

   ii. $m.mlog \in allLogs$

   iii. By the inductive hypothesis, the subterms in $m.mlog$ monotonically grow during its term, so the subterms in $log'[follower]$ monotonically grow during its term.

□

Lemma R.8 is part of the Leader Completeness property in Raft algorithm. Given the existence of shortcut replication, and the fact that witness uses different rules to cast vote, we present proof to the equivalent ***Lemma*** *4 following the same idea as that in the Raft algorithm.*

Lemma 4. *Immediately committed entries are committed*

$$\begin{aligned} &\forall \langle index, term, subterm \rangle \in immediatelyCommitted : \\ &\quad \langle index, term, subterm \rangle \in committed(term) \end{aligned}$$

Proof. 1. Consider an entry $\langle index, term, subterm \rangle$ that is *immediately committed*.

2. Define

$$\begin{aligned} Contradicting \triangleq \{election \in elections : \\ &\wedge election.eterm > term \\ &\wedge \langle index, term, subterm \rangle \notin election.elog\} \end{aligned}$$

3. Let *election* be an element in *Contradicting* with a minimal *term* field. That is,

$$\forall\ e \in Contradicting : election.eterm \leq e.eterm.$$

If more than one election has the same term, choose the earliest one. (The specification does not allow this to happen, but it is safe for a leader to step down and become leader again in the same term)

4. It suffices to show a contradiction, which implies $Contradicting = \phi$.

5. Let $\langle index', term, subterm\rangle$ be any entry that exists in logs of a quorum during $term$, where $index' \leq index$. Such entry must exist if the subterm's replication set contains the witness, since:

   (a) Case $\langle index, term, subterm\rangle$ is immediately committed with no shortcut replication: $\langle index', term, subterm\rangle$ exists in logs of a quorum during $term$, following the specification.

   (b) Case $\langle index, term, subterm\rangle$ is immediately committed with shortcut replication: Following the specification, shortcut replication can only be executed after a log entry in the same subterm being replicated to the witness. Thus $\langle index', term, subterm\rangle$ exists for some $index' < index$.

6. Let $voter$ be either

   - any regular server that both votes in $election$ and contains $\langle index, term, subterm\rangle$ in its log during $term$
   - witness that both votes in $election$ and contains $\langle index', term, subterm\rangle$ in its log during $term$

   Such server must exist, since:

   (a) Case: $\langle index, term, subterm\rangle$ is immediately committed with no shortcut replication.

   $voter$ must exists, since:

      i. $\langle index, term, subterm\rangle$ exists in logs of a quorum during $term$.
      ii. A quorum of servers voted in $election$ to make it succeed.
      iii. Two quorums always overlap.
      iv. $voter$ can be either regular server or witness.

   (b) Case: $\langle index, term, subterm\rangle$ is immediately committed with shortcut replication. $voter$ must exists, since:

      i. A subquorum of regular servers contains $\langle index, term, subterm\rangle$ in its log during $term$. And the witness server contains $\langle index', term, subterm\rangle$ in its log during $term$. These make a quorum.
      ii. A quorum of servers voted in $election$ to make it succeed. The server is either regular server or witness.

iii. Two quorums always overlap.

7. Let $voterLog \triangleq election.evoterLog[voter]$, the voter's log at the time it casts its vote.

8. For any entry $\langle k, term, subterm\rangle (k \leq index)$ that the voter contains during $term$, it is also contained by the voter when it casts its vote during $election.eterm$. That is, $\langle k, term, subterm\rangle \in voterLog$, where $k \leq index$:

   (a) $\langle k, term, subterm\rangle$ was in the $voter$'s log during $term$.

   (b) The $voter$ must have stored the entry in $term$ before voting in $election.eterm$, since:

      i. $election.eterm > term$.

      ii. The $voter$ rejects requests with terms smaller than its current term, and its current term monotonically increases.

   (c) The $voter$ couldn't have removed the entry before casting its vote:

      i. Case: No $AppendEntriesRequest$ with $mterm < term$ removes the entry from the voter's log, since $currentTerm[voter] \geq term$ upon storing the entry, and the voter rejects requests with terms smaller than $currentTerm[voter]$.

      ii. Case: No $AppendEntriesRequest$ with $mterm = term$ removes the entry from the voter's log, since:

         A. There is only one leader of $term$.

         B. The leader of $term$ created and therefore contains the entry.

         C. The leader would not send any conflicting requests to $voter$ during $term$.

      iii. Case: No $AppendEntriesRequest$ with $mterm > term$ removes the entry from the voter's log, since:

         A. Case: $mterm > election.eterm$:
         This can't happen, since $currentTerm[voter] > election.eterm$ would have presented the voter from voting in $term$.

         B. Case: $mterm = election.eterm$:
         Since there is at most one leader per term, this request would have to come from $election.eleader$ as a result of an earlier election in the same term ($election.eterm$).
         Because a leader's log grows monotonically during its term, the leader could not have had $\langle k, term, subterm\rangle$ in its log at

the start of its term, hence it could not have had $\langle index, term, subterm\rangle$ because $index \geq k$. Then there exists an earlier election with the same term in *Contradicting*; this is a contradiction.

C. Case $mterm < election.eterm$:
The leader of *mterm* must have contained $\langle index, term, subterm\rangle$ (otherwise its election would also be *Contradicting* but have a smaller term than *election*, which is a contradiction). Thus, the leader of *mterm* could not send any conflicting entries to the voter for this index, nor could it send any conflicting entries for prior indexes: that it has this entry implies that it has the entrire prefix before it (Raft lemma 4).

9. Thus, we have $\langle index, term, subterm\rangle \in voterLog$ for non-witness voter, and $\langle index', term, subterm\rangle \in voterLog$ for witness voter.

   We show contradiction in following cases which will then be used to show contradiction in each case of voting rules.

10. Case: $LastTerm(election.elog) = LastTerm(voterLog) \wedge Len(election.elog) \geq index$

    (a) The leader of $LastTerm(voterLog)$ monotonically grew its log during its term (by Lemma R.3).

    (b) The same leader must have had $election.elog$ as its log at some point, since it created the last entry.

    (c) Thus, $voterLog[1..index]$ is a prefix of $election.elog$.

    (d) Then $\langle index, term, subterm\rangle \in election.elog$, since $\langle index, term, subterm\rangle \in voterLog[1..index]$.

    (e) But $election \in Contradicting$ implies that $\langle index, term, subterm\rangle \notin election.elog$.

11. Case: $LastTerm(election.elog) > term$

    (a) $election.eterm > LastTerm(election.elog)$ since servers increment their *currentTerm* when starting an election, and Lemma R.6 states that a server's *currentTerm* is at least as large as the terms in its log.

    (b) Let *prior* be the election in *elections* with $prior.eterm = LastTerm(election.elog)$. Such an election must exist since $LastTerm(election.elog) > 0$ and a server must win an election before creating an entry.

(c) By transitivity, we now have the following inequalities:

$$\begin{aligned}
&\mathit{term} \le \\
&\quad \mathit{LastTerm}(\mathit{election.elog}) = \mathit{prior.eterm} < \\
&\qquad \mathit{election.eterm}
\end{aligned}$$

(d) $\langle \mathit{index}, \mathit{term}, \mathit{subterm} \rangle \in \mathit{prior.elog}$, since $\mathit{prior} \notin \mathit{Contradicting}$ ($\mathit{election}$ was assumed to have the lowest term of any election in $\mathit{Contradicting}$, and $\mathit{prior.eterm} < \mathit{election.eterm}$).

(e) $\mathit{prior.elog}$ is a prefix of $\mathit{election.elog}$ since:

  i. $\mathit{prior.eleader}$ creates entries with $\mathit{prior.eterm}$ by appending them to its log, which monotonicallygrows during $\mathit{prior.eterm}$ from $\mathit{prior.elog}$.

  ii. Thus, any entry with term $\mathit{prior.eterm}$ must follow $\mathit{prior.elog}$ in all logs (by Lemma R.4).

  iii. $\mathit{LastTerm}(\mathit{election.elog}) = \mathit{prior.eterm}$

(f) $\langle \mathit{index}, \mathit{term}, \mathit{subterm} \rangle \in \mathit{election.elog}$. Note that this is true no matter $\langle \mathit{index}, \mathit{term}, \mathit{subterm} \rangle$ existing in $\mathit{voterLog}$ or not.

(g) This is a contradiction, since $\mathit{election.elog}$ was assumed to not contain the committed entry ($\mathit{election} \in \mathit{Contradicting}$).

12. The log comparison during elections states the following, since *voter* granted its vote during *election*:

$$\begin{aligned}
&\lor \land \mathit{voter} \neq \mathit{WitnessID} \\
&\quad \land \lor \mathit{LastTerm}(\mathit{election.elog}) > \mathit{LastTerm}(\mathit{voterLog}) \\
&\qquad \lor \land \mathit{LastTerm}(\mathit{election.elog}) = \mathit{LastTerm}(\mathit{voterLog}) \\
&\qquad\quad \land \mathit{Len}(\mathit{election.elog}) \ge \mathit{Len}(\mathit{voterLog}) \\
&\lor \land \mathit{voter} = \mathit{WitnessID} \\
&\quad \land \lor \mathit{LastTerm}(\mathit{election.elog}) > \mathit{LastTerm}(\mathit{voterLog}) \\
&\qquad \lor \land \mathit{LastTerm}(\mathit{election.elog}) = \mathit{LastTerm}(\mathit{voterLog}) \\
&\qquad\quad \land \mathit{LastSubterm}(\mathit{election.elog}) > \mathit{LastSubterm}(\mathit{voterLog}) \\
&\qquad \lor \land \mathit{LastTerm}(\mathit{election.elog}) = \mathit{LastTerm}(\mathit{voterLog}) \\
&\qquad\quad \land \mathit{LastSubterm}(\mathit{election.elog}) = \mathit{LastSubterm}(\mathit{voterLog}) \\
&\qquad\quad \land \mathit{election.evoters} \setminus \mathit{WitnessID} \subset \mathit{ReplicationSet}
\end{aligned}$$

$ReplicationSet$ is replication set of $LastSubterm(voterLog)$ in $LastTerm(voterLog)$ if $voter$ is witness.

We now investigate each voting rule in following cases, and use the result of above two cases to show contradiction.

13. Case: $voter$ is a regular server, and $LastTerm(election.elog) = LastTerm(voterLog) \wedge Len(election.elog) \geq Len(voterLog)$
    (a) $Len(election.elog) \geq Len(voterLog) \implies Len(election.elog) \geq index$
    (b) This leads to contradiction following case 10
14. Case: $LastTerm(election.elog) > LastTerm(voterLog)$, no matter $voter$ being a regular server or witness
    (a) $LastTerm(election.elog) > LastTerm(voterLog) \implies LastTerm(election.elog) > term$
    (b) This leads to contradiction following case 11
15. Case: $voter$ is witness, and $LastTerm(election.elog) = LastTerm(voterLog) \wedge LastSubterm(election.elog) > LastSubterm(voterLog)$
    (a) $LastSubterm(election.elog) > LastSubterm(voterLog) \implies Len(election.elog) > Len(voterLog) > term$, following Lemma 3
    (b) This leads to contradiction following case 10
16. Case: $voter$ is witness, and

$$\begin{aligned}
&\wedge LastTerm(election.elog) = LastTerm(voterLog)\\
&\wedge LastSubterm(election.elog) = LastSubterm(voterLog)\\
&\wedge election.evoters \cup WitnessID \subset ReplicationSet
\end{aligned}$$

    (a) $LastTerm(voterLog) = term \wedge LastSubterm(voterLog) = subterm$
        i. There must be $voter'$ which is a regular server that contains $\langle index, term, subterm\rangle$ and votes in $election$, since
            A. $\langle index, term, subterm\rangle$ exists in at least subquorum regular servers in replication set of $term$ and $subterm$, no matter if it is shortcut replicated or not.
            B. A subquorum in replication set of $term$ and $subterm$ vote in $election$.

C. Subquorums of regular servers overlap in a replication set.

ii. This leads to contradiction following case 13 and 14

(b) $LastTerm(voterLog) = term \wedge LastSubterm(voterLog) > subterm$

i. $LastTerm(voterLog) = term \wedge LastSubterm(voterLog) > subterm \implies Len(voterLog) > index$, per Lemma 3

ii. This leads to contradiction following case 10

(c) $LastTerm(voterLog) > term$. This leads to contradiction following case 11

Having Lemma 4 proved, Lemma R.9 and Theorem R.1 are true, following the same proof in Raft algorithm. Now we have Leader Completeness and State Machine Safety proved for the extended Raft algorithm. □