Consensus Bridging Theory And Practice

wu

July 1, 2024

1 Motivation

1.1 Achieving fault tolerance with replicated state machines

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, they are said to be **committed**. Each server's state machine processes committed commands in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

2 Basic Raft algorithm

2.1 Raft overview

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§3.5); also used as heartbeat (§3.4).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§3.4).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§3.6)
lastLogTerm	term of candidate's last log entry (§3.6)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§3.5)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§3.3)

Followers (§3.4):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§3.4):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§3.4)
- If command received from client: append entry to local log, respond after entry applied to state machine (§3.5)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§3.5)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§3.5, §3.6).

Figure 3.1: A condensed summary of the Raft consensus algorithm (excluding membership changes, log compaction, and client interaction). The server behavior in the lower-right box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §3.4 indicate where particular features are discussed. The formal specification in Appendix B describes the algorithm more precisely.

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- Leader election: a new leader must be chosen when starting the cluster and when an existing leader fails
- Log replication: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own
- Safety: the key safety property for Raft is the State Machine Safety Property

Raft **SAFETY**:

- **Election Safty**: At most one leader can be elected in a given term.
- **Leader Append-Only**: A leader never overwrites or deletes entries in its log; it only appends new entries.
- **Log Matching**: If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
- **Leader Completeness**: If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
- **State Machine Safety**: If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

2.2 Log replication

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called **committed**.

- Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines.
- A log entry is committed once the leader that created the entry has replicated it on a majority of the servers. This also commits all preceding entries in the leader's log, including entries created by previous leaders.

- The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out.
- Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).

Raft maintains the following properties, which together constitute the **Log Matching Property**:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.
- The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log.

  If two entries have the same term, then they come from the same leader. If a log is replicated into a specific entry, then the index of that log is the same as the leader. Therefore the two entries have the same command as they come from the same entry from the same leader in the same term.
- The second property is guaranteed by a consistency check performed by AppendEntries. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries(prev log). If the follower does not find an entry in its log with the same index and term, then it refuses the new entries.

  The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

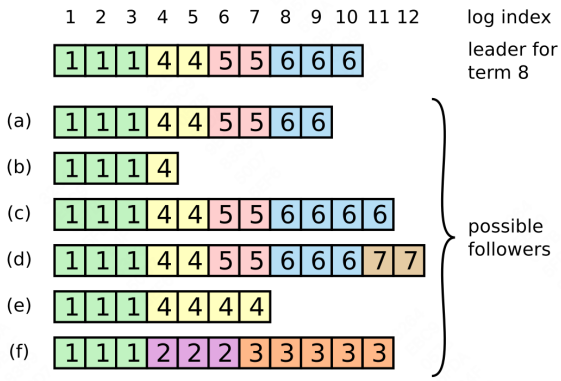


Figure 3.6: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

The leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs:

- The leader maintains a **nextIndex** for each follower, which is the index of the next log entry the leader will send to that follower.
- When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log.
- If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements the follower's nextIndex and retries the AppendEntries RPC. Eventually the nextIndex will reach a point where the leader and follower logs match.

Until the leader has discovered where it and the follower's logs match, the leader can send AppendEntries with no entries (like heartbeats) to save

bandwidth. Then, once the matchIndex immediately precedes the nextIndex, the leader should begin to send the actual entries.

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs:

- when rejecting an AppendEntries request, the follower can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term;
- the leader can use a binary search approach to find the first entry where the follower's log differs from its own; this has better worst-case behavior.

2.3 Safty

This section completes the Raft algorithm by adding a restriction on which servers may be elected leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms.

2.3.1 Election restriction

In any leader-based consensus algorithm, the leader **must** eventually store all of the committed log entries.

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries:

- A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers.
- If the candidate's log is at least as **up-to-date** as any other log in that majority, then it will hold all the committed entries.

Raft determines which of two logs is more **up-to-date** by comparing the index and term of the last entries in the logs.

- If the logs have last entries with different terms, then the log with the later term is more up-to-date.
- If the logs end with the same term, then whichever log is longer is more up-to-date.

The correctness of this notion of up-to-date comes from Log Matching Property.

2.3.2 Committing entries from previous terms

A leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers:

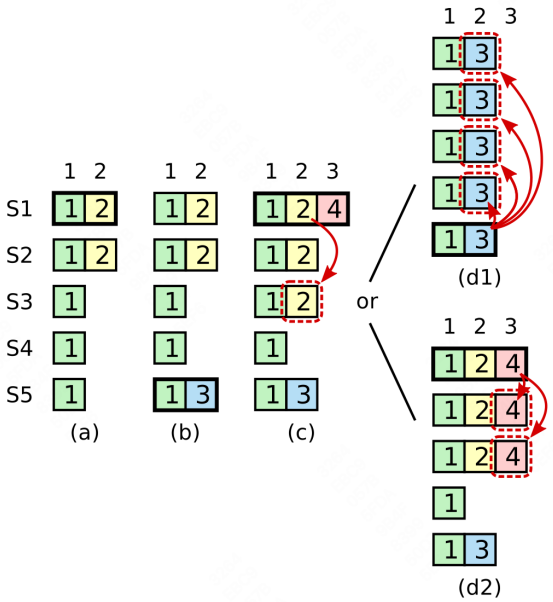


Figure 3.7: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d1), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (d2), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

To eliminate problems like the one in Figure 2.3.2, Raft never commits log entries from previous terms by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property.

2.3.3 Safety argument

Assume Leader Completeness Property does not hold. Suppose the leader for term T $leader_T$ commits a log entry from its term, but that log entry is

not stored by the leader of some future term. Consider the smallest term $U > T$ whose leader $leader_U$ does not store the entry.

1. The committed entry must have been absent from $leader_U$'s log at the time of its election.

2. $leader_T$ replicated the entry on a majority of the cluster, and $leader_U$ received votes from a majority of the cluster. Thus at least one server both accepted the entry from $leader_T$ and voted for $leader_U$.

3. The voter must have accepted the committed entry from $leader_T$ **before** voting for $leader_U$; otherwise it would have rejected the AppendEntries request from $leader_T$

4. The voter still stored the entry when it voted for $leader_U$, since every intervening leader contained the entry, leaders never remove entries, and followers only remove entries if they conflict with the leader.

5. The voter granted its vote to $leader_U$, so $leader_U$'s log must have been as up-to-date as the voter's. This leaders to one of two contradictions.

6. First, if the voter and $leader_U$ shared the same last log term, then $leader_U$'s log must have been at least as long as the voter's, so its log contained every entry in the voter's log.

7. Otherwise, $leader_U$'s last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T. The earlier leader that created $leader_U$'s last log entry must have contained the committed entry in its log. Then by the Log Matching Property, $leader_U$'s log must also contain the committed entry, which is a contradiction.

8. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.

9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly.

Given the Leader Completeness Property, we can prove the State Machine Safety Property from, which states that if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index:

- At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry, and the entry must be committed. Now consider the lowest term in which any server applies a given log index; the Leader Completeness Property guarantees that the leaders for all higher terms will store that same log entry, so servers that apply the index in later terms will apply the same value. Thus, the State Machine Safety Property holds.

Finally, Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

2.3.4 Followe rand candidate crashes

2.3.5 Persisted state and server restarts

- current term and vote: prevent the server from voting twice means vote for different candidates in the same term or replacing log entries from a newer leader with those from a deposed leaderterm.
- new log entries before they are committed: prevents committed entries from being lost or "uncommitted" when servers restart.
- *last applied* index

2.3.6 Timing and availability

2.3.7 Leadership transfer extention

To transfer leadership in Raft, the prior leader sends its log entries to the target server, then the target server runs an election without waiting for the election timeout to elaspe.

1. The prior leader stops accepting new client requests.
2. The prior leader fully updates the target server's log to match its own, using the normal log replication mechanism
3. The prior leader sends a *TimeoutNow* request to the target server.

Once the target server receives the TimeoutNow request, it is highly likely to start an election before any other server and become leader in the next term. Its next message to the prior leader will include its new term

number, causing the prior leader to step down. At this point, leadership transfer is complete.

It is also possible for the target server to fail; in this case, the cluster must resume client operations. If leadership transfer does not complete after about an election timeout, the prior leader aborts the transfer and resumes accepting client requests. If the prior leader was mistaken and the target server is actually operational, then at worst this mistake will result in an extra election, after which client operations will be restored.

3 Cluster membership changes

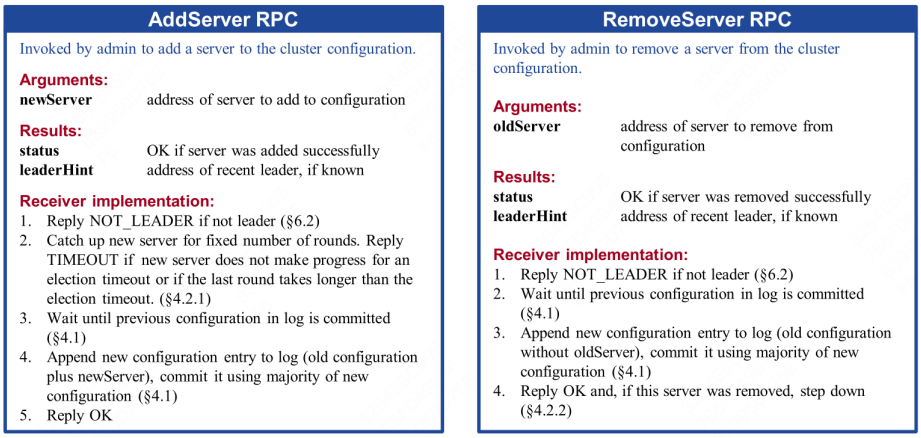


Figure 4.1: RPCs used to change cluster membership. The AddServer RPC is used to add a new server to the current configuration, and the RemoveServer RPC is used to remove a server from the current configuration. Section numbers such as §4.1 indicate where particular features are discussed. Section 4.4 discusses ways to use these RPCs in a complete system.

3.1 Safety

- Goal: no point during the transition where it is possible for two leaders to be elected for the same term.
- Difficulty: it isn't possible to atomically switch all of the servers at once, so the cluster can potentially split into two independent majorities during the transition

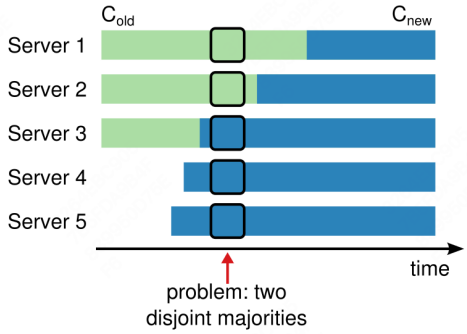
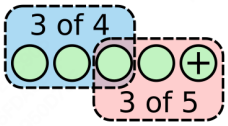


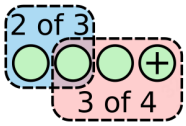
Figure 4.2: Switching directly from one configuration to another can be unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

Raft restricts the types of changes that are allowed: only one server can be added or removed from the cluster at a time. More complex changes in membership are implemented as a series of single-server changes.

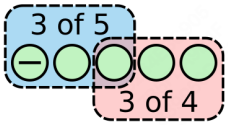
When adding a single server to a cluster or removing a single server from a cluster, any majority of the old cluster overlaps with any majority of the new cluster. This overlap prevents the cluster from splitting into two independent majorities; in terms of the safety argument of Section 2.3.3, it guarantees the existence of "the voter". Thus, when adding or removing just a single server, it is safe to switch directly to the new configuration.



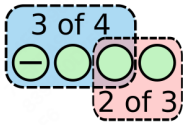
(a) Adding one server to a 4-server cluster.



(b) Adding one server to a 3-server cluster.



(c) Removing one server from a 5-server cluster.



(d) Removing one server from a 4-server cluster.

Figure 4.3: The addition and removal of a single server from an even- and an odd-sized cluster. In each figure, the blue rectangle shows a majority of the old cluster, and the red rectangle shows a majority of the new cluster. In every single-server membership change, an overlap between any majority of the old cluster and any majority of the new cluster is preserved, as needed for safety. For example in (b), a majority of the old cluster must include two of the left three servers, and a majority of the new cluster must include three of the servers in the new cluster, of which at least two must come from the old cluster.

When the leader receives a request to add or remove a server from its current configuration C_{old}, it appends the new configuration C_{new} as an entry in its log and replicates that entry using the normal Raft mechanism. The new configuration takes effect on each server as soon as it is added to that server's log: the C_{new} entry is replicated to the C_{new} servers, and a majority of the new configuration is used to determine the C_{new} entry's commitment.

The configuration change is complete once the C_{new} entry is committed. At this point, the leader knows that a majority of the C_{new} servers have adopted C_{new} . It also knows that any servers that have not moved to C_{new} can no longer form a majority of the cluster, and servers without C_{new} cannot be elected leader. Commitment of C_{new} allows three things to continue:

1. The leader can acknowledge the successful completion of the configuration change.

2. If the configuration change removed a server, that server can be shut down.

3. Further configuration changes can be started. Before this point, overlapped configuration changes could degrade to unsafe situations as in Fig 3.1

- Servers always use the latest configuration in their logs, regardless of whether that configuration entry has been committed. This allows leaders to easily avoid overlapping configuration changes (the third item above), by not beginning a new change until the previous change's entry has committed.

- It is only safe to start another membership change once a majority of the old cluster has moved to operating under the rules of C_{new}. If servers adopted C_{new} only when they learned that C_{new} was committed, Raft leaders would have a difficult time knowing when a majority of the old cluster had adopted it.

In Raft, it is the caller's configuration that is used in reaching consensus, both for voting and for log replication:

- A server accepts `AppendEntries` requests from a leader that is not part of the server's latest configuration. Otherwise, a new server could never be added to the cluster (it would never accept any log entries preceding the configuration entry that adds the server).

- A server also grants its vote to a candidate that is not part of the server's latest configuration (if the candidate has a sufficiently up-to-date log and a current term). This vote may occasionally be needed to keep the cluster available. For example, consider adding a fourth server to a three-server cluster. If one server were to fail, the new server's vote would be needed to form a majority and elect a leader.

Thus, servers process incoming RPC requests without consulting their current configurations.

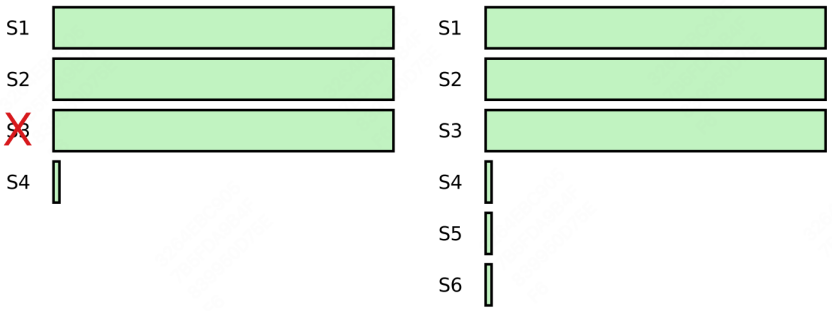
3.2 Availability

3.2.1 Catching up new servers

When a server is added to the cluster, it typically will not store any log entries. If it is added to the cluster in this state, its log could take quite a while to catch up to the leader's, and during this time, the cluster is more vulnerable to unavailability:

- A three-server cluster can normally tolerate one failure with no loss in availability. However, if a fourth server with an empty log is added to the same cluster and one of the original three servers fails, the cluster will be temporarily unable to commit new entries (3.2.1 (a)).

- Many new servers are added to a cluster in quick succession, where the new servers are needed to form a majority of the cluster (3.2.1 (b)).



(a) Failure of S3 while adding S4.

(b) Adding S3–S6 in quick succession.

Figure 4.4: Examples of how adding servers with empty logs can put availability at risk. The figures show the servers' logs in two different clusters. Each cluster starts out with three servers, S1–S3. In (a), S4 is added, then S3 fails. The cluster should be able to operate normally after one failure, but it loses availability: it needs three of the four servers to commit a new entry, but S3 has failed and S4's log is too far behind to append new entries. In (b), S4–S6 are added in quick succession. Committing the configuration entry that adds S6 (the third new server) requires four servers' logs to store that entry, but S4–S6 have logs that are far behind. Neither cluster will be available until the new servers' logs are caught up.

In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which a new server joins the cluster as a **non-voting member**. The leader replicates log entries to it, but it is not yet counted towards majorities for voting or commitment purposes. Once the new server has caught up with the rest of the cluster, the reconfiguration can proceed as described above. (The mechanism to support non-voting servers can also be useful in other contexts; for example, it can be used to replicate the state to a large number of servers, which can serve read-only requests with relaxed consistency.)

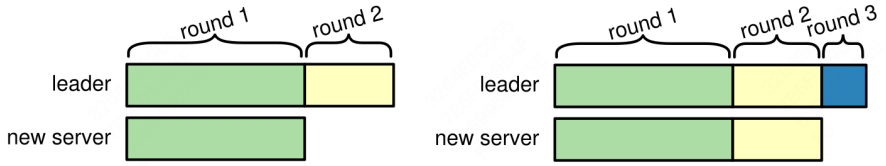
We suggest the following algorithm to determine when a new server is sufficiently caught up to add to the cluster:

- The replication of entries to the new server is split into rounds, as shown in Figure 3.2.1.

- Each round replicates all the log entries present in the leader's log at the start of the round to the new server's log. While it is replicating entries for its current round, new entries may arrive at the leader; it

will replicate these during the next round. As progress is made, the round durations shrink in time.

- The algorithm waits a fixed number of rounds (such as 10). If the last round lasts less than an election timeout, then the leader adds the new server to the cluster, under the assumption that there are not enough unreplicated entries to create a significant availability gap.
- Otherwise, the leader aborts the configuration change with an error. The caller may always try again (it will be more likely to succeed the next time, since the new server's log will already be partially caught up).



(a) Start of round 2. **(b)** End of round 2.

Figure 4.5: To catch up a new server, the replication of entries to the new server is split into rounds. Each round completes once the new server has all of the entries that the leader had in its log at the start of the round. By then, however, the leader may have received new entries; these are replicated in the next round.

3.2.2 Removing the current leader

3.2.3 Disruptive servers

Without additional mechanism, servers not in C_{new} can disrupt the cluster.

- Once the cluster leader has created the C_{new} entry, a server that is not in C_{new} will no longer receive heartbeats, so it will time out and start new elections.
- Furthermore, it will not receive the C_{new} entry or learn of that entry's commitment, so it will not know that it has been removed from the cluster. The server will send `RequestVote` RPCs with new term numbers, and this will cause the current leader to revert to follower state.
- A new leader from C_{new} will eventually be elected, but the disruptive server will time out again and the process will repeat, resulting in poor availability. If multiple servers have been removed from the cluster, the situation could degrade further.

First idea was that, if a server is going to start an election, it would first check that it wouldn't be wasting everyone's time - that it had a chance to win the election. This introduced a new phase to elections, called the **Pre-Vote phase**. A candidate would first ask other servers whether its log was up-to-date enough to get their vote. Only if the candidate believed it could get votes from a majority of the cluster would it increment its term and start a normal election.

Unfortunately, the Pre-Vote phase does not solve the problem of disruptive servers: there are situations where the disruptive server's log is sufficiently up-to-date, but starting an election would still be disruptive. Perhaps surprisingly, these can happen even before the configuration change completes.

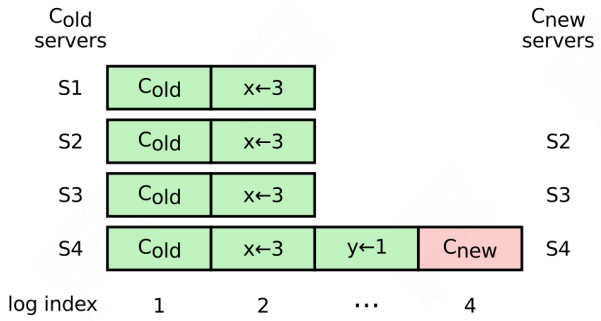


Figure 4.7: An example of how a server can be disruptive even before the C_{new} log entry has been committed, and the Pre-Vote phase doesn't help. The figure shows the removal of S1 from a four-server cluster. S4 is leader of the new cluster and has created the C_{new} entry in its log, but it hasn't yet replicated that entry. Servers in the old cluster no longer receive heartbeats from S4. Even before C_{new} is committed, S1 can time out, increment its term, and send this larger term number to the new cluster, forcing S4 to step down. The Pre-Vote algorithm does not help, since S1's log is as up-to-date as a majority of either cluster.

Raft's solution uses heartbeats to determine when a valid leader exists. We modify the `RequestVote` RPC to achieve this: if a server receives a `RequestVote` request within the minimum election timeout of hearing from a current leader, it does not update its term or grant its vote. It can either drop the request, reply with a vote denial, or delay the request; the result is essentially the same. This does not affect normal elections, where each server waits at least a minimum election timeout before starting an election. However, it helps avoid disruptions from servers not in C_{new} : while a leader is able to get heartbeats to its cluster, it will not be deposed by larger term numbers.

3.2.4 Availability

We show that the algorithm will be able to maintain and replace leaders during membership changes and that the leader(s) will both service client requests and complete the configuration changes. We assume, among other things, that a majority of the old configuration is available (at least until C_{new} is committed) and that a majority of the new configuration is available.

1. A leader can be elected at all steps of the configuration change:
    - If the available server with the most up-to-date log in the new cluster has the C_{new} entry, it can collect votes from a majority of C_{new} and become leader
    - Otherwise, the C_{new} entry must not yet be committed. The available server with the most up-to-date log among both the old and new clusters can collect votes from a majority of C_{old} and a majority of C_{new}, so no matter which configuration it uses, it can become leader.

3.3 Arbitrary configuration changes using joint consensus

To ensure safety across arbitrary configuration changes, the cluster first switches to a transitional configuration we call **joint consensus**; once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:

- Log entries are replicated to all servers in both configurations
- Any server from either configuration may serve as leader
- Agreement (for elections and entry commitment) requires separate majorities from **both** the old and new configurations.

The joint consensus allows individual servers to transition between configurations at different times without compromising safety. Furthermore, joint consensus allows the cluster to continue servicing client requests throughout the configuration change.

This approach extends the single-server membership change algorithm with an intermediate log entry for the joint configuration; Figure 4.8 illustrates the process.

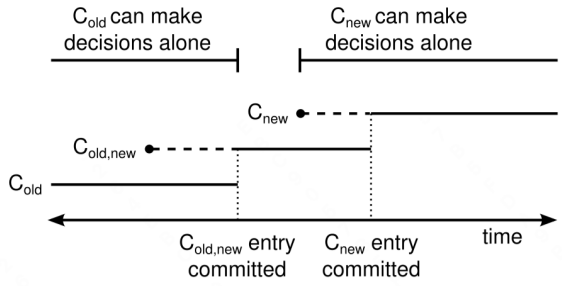


Figure 4.8: Timeline for a configuration change using joint consensus. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{\text{old,new}}$ configuration entry in its log and commits it to $C_{\text{old,new}}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

- When the leader receives a request to change the configuration from C_{old} to C_{new}, it stores the configuration for joint consensus ($C_{old,new}$ in the figure) as a log entry and replicates that entry using the normal Raft mechanism.

- As with the single-server configuration change algorithm, each server starts using a new configuration as soon as it stores the configuration in its log. This means that the leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed. If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$

- Once $C_{old,new}$ has been committed, neither C_{old} and C_{new} can make decisions without approval of the other, and the Leader Completeness Property ensures that only servers with the $C_{old,new}$ log entry can be elected as leader. It is now safe for the leader to create a log entry describing C_{new} and replicate it to the cluster. Again this configuration will take effect on each server as soon as it is seens. When the C_{new} log entry has been committed under the rules of C_{new} , the old configuration is irrelevant and servers not in the new configuration can be shut down.

4 Log compaction

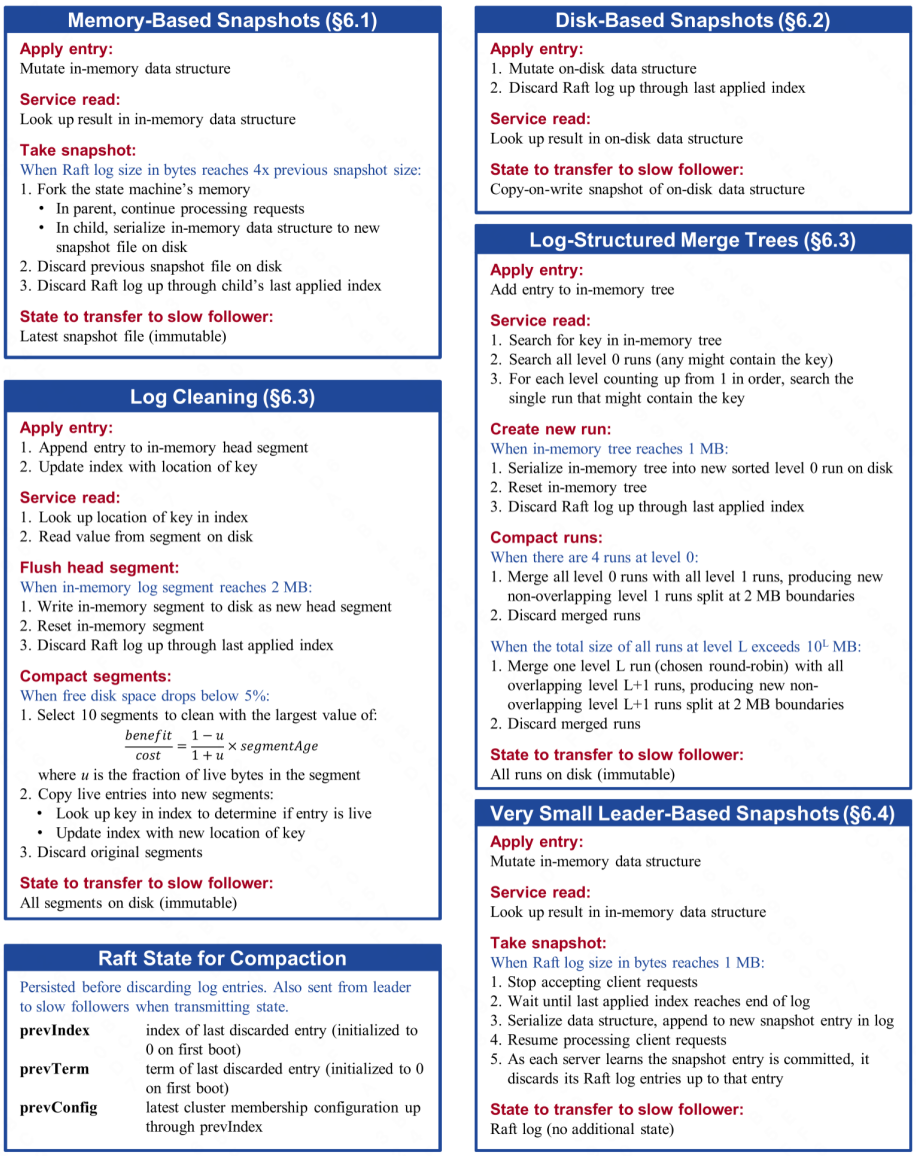


Figure 5.1: The figure shows how various approaches to log compaction can be used in Raft. Details for log-structured merge trees in the figure are based on LevelDB [63], and details for log cleaning are based on RAMCloud [98]; rules for managing deletions are omitted.

The various approaches to compaction share several core concepts.

- Instead of centralizing compaction decisions on the leader, each server compacts the committed prefix of its log independently.
- The basic interaction between the state machine and Raft involves transferring responsibility for a prefix of the log from Raft to the state machine.
- Once Raft has discarded a prefix of the log, the state machine takes on two new responsibilities.

    1. If the server restarts, the state machine will need to load the state corresponding to the discarded log entries from disk before it can apply any entries from the Raft log.
    2. the state machine may need to produce a consistent image of the state so that it can be sent to a slow follower.

    It is not feasible to defer compaction until log entries have been "fully replicated" to every member in the cluster, since a minority of slow followers must not keep the cluster from being fully available, and new servers can be added to the cluster at any time.

4.1 Snapshotting memory-based state machines

Each server takes snapshots independently, covering just the committed entries in its log. Most of the work in snapshotting involves serializing the state machine's current state, and this is specific to a particular state machine implementation.

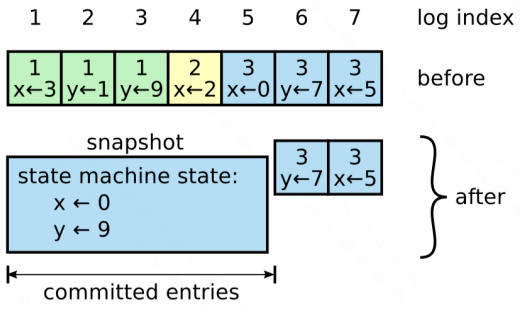


Figure 5.2: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables x and y in this example). Before discarding entries 1 though 5, Raft saves the snapshot's last included index (5) and term (3) to position the snapshot in the log preceding entry 6.

Once the state machine completes writing a snapshot, the log can be truncated. Raft first stores the state it needs for a restart:

- the index and term of the last entry included in the snapshot
- the latest configuration as of that index.

Then it discards the prefix of its log up through that index. Any previous snapshots can also be discarded, as they are no longer useful.

The leader may occasionally need to send its state to slow followers and to new servers that are joining the cluster. In snapshotting, this state is just the latest snapshot, which the leader transfers using a new RPC called `InstallSnapshot`:

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower.
Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIndex	the snapshot replaces all entries up through and including this index
lastTerm	term of lastIndex
lastConfig	latest cluster configuration as of lastIndex (include only with first chunk)
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. If lastIndex is larger than latest snapshot's, save snapshot file and Raft state (lastIndex, lastTerm, lastConfig). Discard any existing or partial snapshot.
6. If existing log entry has same index and term as lastIndex and lastTerm, discard log up through lastIndex (but retain any following entries) and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load lastConfig as cluster configuration)

Figure 5.3: Leaders invoke the InstallSnapshot RPC to send snapshots to slow followers. Leaders resort to sending a snapshot only when they have already discarded the next log entry needed to replicate entries to the follower with AppendEntries. They split the snapshot into chunks for transmission. Among other benefits, this gives the follower a sign of life with each chunk, so it can reset its election timer. Each chunk is sent in order, which simplifies writing the file to disk. The RPC includes the state needed for Raft to load the snapshot on a restart: the index and term of the last entry covered by the snapshot, and the latest configuration at that point.

When a follower receives a snapshot with this RPC, it must decide what to do with its existing log entries.

- Usually the snapshot will contain new information not already in the follower's log. In this case, the follower discards its entire log; it is all superseded by the snapshot and may possibly have uncommitted entries that conflict with the snapshot.

- If the follower receives a snapshot that describes a prefix of its log (due to retransmission or by mistake), then log entries covered by the snapshot are deleted but entries following the snapshot are still valid and must be retained.

4.1.1 Snapshotting concurrently

Creating a snapshot can take a long time, both in serializing the state and in writing it to disk. Thus, both serializing and writing snapshots must be concurrent with normal operations to avoid availability gaps.

Fortunately, copy-on-write techniques allow new updates to be applied without impacting the snapshot being written. There are two approaches to this:

- State machines can be built with immutable (functional) data structures to support this. Because state machine commands would not modify the state in place, a snapshotting task could keep a reference to some prior state and write it consistently into a snapshot.

• Alternatively, the operating system's copy-on-write support can be used (where the programming environment allows it). On Linux for example, in-memory state machines can use fork to make a copy of the server's entire address space. Then, the child process can write out the state machine's state and exit, all while the parent process continues servicing requests. The LogCabin implementation currently uses this approach.

4.1.2 When to snapshot

Fortunately, using the size of the **previous** snapshot rather than the size of the next one results in reasonable behavior.

4.1.3 Implementation concerns

- **Saving and loading snapshots**:

4.2 Snapshotting disk-based state machines

Disk-based state machines must be able to provide a consistent snapshot of the disk for the purpose of transmitting it to slow followers.

5 Client Interaction

ClientRequest RPC

Invoked by clients to modify the replicated state.

Arguments:

clientId	client invoking request (§6.3)
sequenceNum	to eliminate duplicates (§6.4)
command	request for state machine, may affect state

Results:

status	OK if state machine applied command
response	state machine output, if successful
leaderHint	address of recent leader, if known (§6.2)

Receiver implementation:

1. Reply NOT_LEADER if not leader, providing hint when available (§6.2)
2. Append command to log, replicate and commit it
3. Reply SESSION_EXPIRED if no record of clientId or if response for client's sequenceNum already discarded (§6.3)
4. If sequenceNum already processed from client, reply OK with stored response (§6.3)
5. Apply command in log order
6. Save state machine output with sequenceNum for client, discard any prior response for client (§6.3)
7. Reply OK with state machine output

Rules for Leaders

- Upon becoming leader, append *no-op* entry to log (§6.4)
- If election timeout elapses without successful round of heartbeats to majority of servers, convert to follower (§6.2)

RegisterClient RPC

Invoked by new clients to open new session, used to eliminate duplicate requests. §6.3

No arguments

Results:

status	OK if state machine registered client
clientId	unique identifier for client session
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader, providing hint when available (§6.2)
2. Append register command to log, replicate and commit it
3. Apply command in log order, allocating session for new client
4. Reply OK with unique client identifier (the log index of this register command can be used)

ClientQuery RPC

Invoked by clients to query the replicated state (read-only commands). §6.4

Arguments:

query	request for state machine, read-only

Results:

status	OK if state machine processed query
response	state machine output, if successful
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader, providing hint when available (§6.2)
2. Wait until last committed entry is from this leader's term
3. Save commitIndex as local variable readIndex (used below)
4. Send new round of heartbeats, and wait for reply from majority of servers
5. Wait for state machine to advance at least to the readIndex log entry
6. Process query
7. Reply OK with state machine output

Figure 6.1: Clients invoke the ClientRequest RPC to modify the replicated state; they invoke the ClientQuery RPC to query the replicated state. New clients receive their client identifier using a RegisterClient RPC, which helps identify when session information needed for linearizability has been discarded. In the figure, servers that are not leaders redirect clients to the leader, and read-only requests are serviced without relying on clocks for linearizability (the text presents alternatives). Section numbers such as §6.3 indicate where particular features are discussed.

5.1 Finding the cluster

Two general approaches:

5.2 Routing requests to the leader

5.3 Implementing linearizable semantics

Raft provides at-least-once semantics for clients; the replicated state machine may apply a command multiple times.

In linearizability, each operation appears to execute instantaneously, exactly once, at some point between its invocation and its response.

To achieve linearizability in Raft, servers must filter out duplicate requests. The basic idea is that servers save the results of client operations and use them to skip executing the same request multiple times. To implement this, each client is given a unique identifier, and clients assign unique serial numbers to every command.

Given this filtering of duplicate requests, Raft provides linearizability. The Raft log provides a serial order in which commands are applied on every server. Commands take effect instantaneously and exactly once according to their first appearance in the Raft log, since any subsequent appearances are filtered out by the state machines as described above.

5.4 Processing read-only queries more efficiently

Fortunately, it is possible to bypass the Raft log for read-only queries and still preserve linearizability. To do so, the leader takes the following steps:

1. If the leader has not yet marked an entry from its current term committed, it waits until it has done so. The Leader Completeness Property guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank **no-op** entry into the log at the start of its term. As soon as this no-op entry is committed, the leader's commit index will be at least as large as any other servers' during its term.

2. The leader saves its current commit index in a local variable `readIndex`.

3. The leader needs to make sure it hasn't been superseded by a newer leader of which it is unaware. It issues a new round of heartbeats and waits for their acknowledgments from a majority of the cluster. Once

these acknowledgments are received, the leader knows that there could not have existed a leader for a greater term at the moment it sent the heartbeats. Thus, the `readIndex` was, at the time, the largest commit index ever seen by any server in the cluster {all later leaders have the same log before `readIndex` thanks to Leader Completeness Property}.

4. The leader waits for its state machine to advance at least as far as the `readIndex`; this is current enough to satisfy linearizability.

5. The leader issues the query against its state machine and replies to the client with the results

To improve efficiency further, the leader can amortize the cost of confirming its leadership: it can use a single round of heartbeats for any number of read-only queries that it has accumulated.

Followers could also help offload the processing of read-only queries. However, these reads would also run the risk of returning stale data without additional precautions. To serve reads safely, the follower could issue a request to the leader that just asked for a current `readIndex` (the leader would execute steps 1–3 above); the follower could then execute steps 4 and 5 on its own state machine for any number of accumulated read-only queries.

5.4.1 Using clocks to reduce messaging for read-only queries

To use clocks instead of messages for read-only queries, the normal heartbeat mechanism would provide a form of lease. Once the leader's heartbeats were acknowledged by a majority of the cluster, the leader would assume that no other server will become leader for about an election timeout, and it could extend its lease accordingly. The leader would then reply to read-only queries during that period without any additional communication. (The leadership transfer mechanism presented in Chapter 3 allows the leader to be replaced early; a leader would need to expire its lease before transferring leadership.)

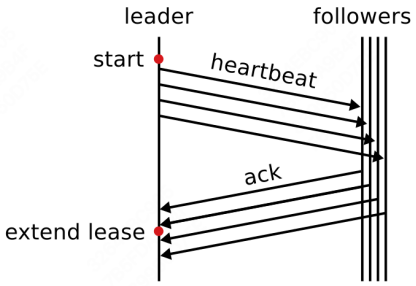


Figure 6.3: To use clocks instead of messages for read-only queries, the leader would use the normal heartbeat mechanism to maintain a lease. Once the leader's heartbeats were acknowledged by a majority of the cluster, it would extends its lease to $start + \frac{election\ timeout}{clock\ drift\ bound}$, since the followers shouldn't time out before then. While the leader held its lease, it would service read-only queries without communication.

The lease approach assumes a bound on clock drift across servers (over a given time period, no server's clock increases more than this bound times any other). Discovering and maintaining this bound might present operational challenges.

Fortunately, a simple extension can improve the guarantee provided to clients, so that even under asynchronous assumptions (even if clocks were to misbehave), each client would see the replicated state machine progress monotonically (sequential consistency). For example, a client would not see the state as of log index n, then change to a different server and see only the state as of log index $n - 1$.

- To implement this guarantee, servers would include the index corresponding to the state machine state with each reply to clients. Clients would track the latest index corresponding to results they had seen, and they would provide this information to servers on each request. If a server received a request for a client that had seen an index greater than the server's last applied log index, it would not service the request (yet).

6 Correctness

6.1 Formal specification and proof for basic Raft algorithm

The specification models an asynchronous system (it has no notion of time) with the following assumptions:

- Messages may take an arbitrary number of steps (transitions) to arrive at a server. Sending a message enables a transition to occur (the receipt of the message) but with no particular timeliness.
- Servers fail by stopping and may later restart from stable storage on disk.
- The network may reorder, drop, and duplicate messages.

7 Leader election evaluation

7.1 Preventing disruptions when a server rejoins the cluster

One downside of Raft's leader election algorithm is that a server that has been partitioned from the cluster is likely to cause a disruption when it regains connectivity.

In the Pre-Vote algorithm, a candidate only increments its term if it first learns from a majority of the cluster that they would be willing to grant the candidate their votes:

1. if the candidate's log is sufficiently up-to-date
2. voters have not received heartbeats from a valid leader for at least a baseline election timeout

8 Problem

1. 3.2.3: Not quite understand
2. 3.2.4: same roblem