Raft: In Search of an Understandable Consensus Algorithm

Link to paper

Introduction

Why Raft at all? Kubernetes uses etcd as its backing data store, and etcd uses Raft as its consensus algorithm!

Consensus is needed to form a system of **replicated state machines**. A typical implementation might be a replicated log - a series of immutable append-only events can be processed in order by each state machine to deterministically compute a final state. With the help of a consensus algorithm, all state machines will agree on the final state at all times, even if a particular machine is down or otherwise experiences trouble.

Consensus algorithms must satisfy the following properties:

1. **Safe** under non-Byzantine conditions, including partitions, packet reordering / delays / loss, and duplication.
2. **Available** given a quorum of healthy nodes.
3. They do **NOT** depend on **clocks**. Faulty clocks are terrible. Google tackled this with Spanner's TrueTime API by building error into their time API, and having massive redundancies to synchronize their time error bounds.
4. Commands can complete using a subset of machines - "a minority of slow servers need not impact overall system performance".

Raft was created in light of Paxos for two main reasons:

1. Paxos is hard to understand. Single-decree Paxos is hard - multi- harder.
2. Implementation details are mostly elided from Lamport's description (almost completely in the case of multi-Paxos). Complexity-reducing optimizations such as leader election and subsequent coordination are only lightly highlighted (Paxos is symmetrically peer-to-peer -- this isn't terribly needed for implementations in the wild).

These two lead to comments like this (from Chubby):

> There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. . . . the final system will be based on an unproven protocol

Because of all this, Raft emphasizes **understandability** above all else. That means that, given two competing algorithmic choices, the more understandable one would win:

> how hard is it to explain each alternative (for example, how complex is its state space, and does it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications?

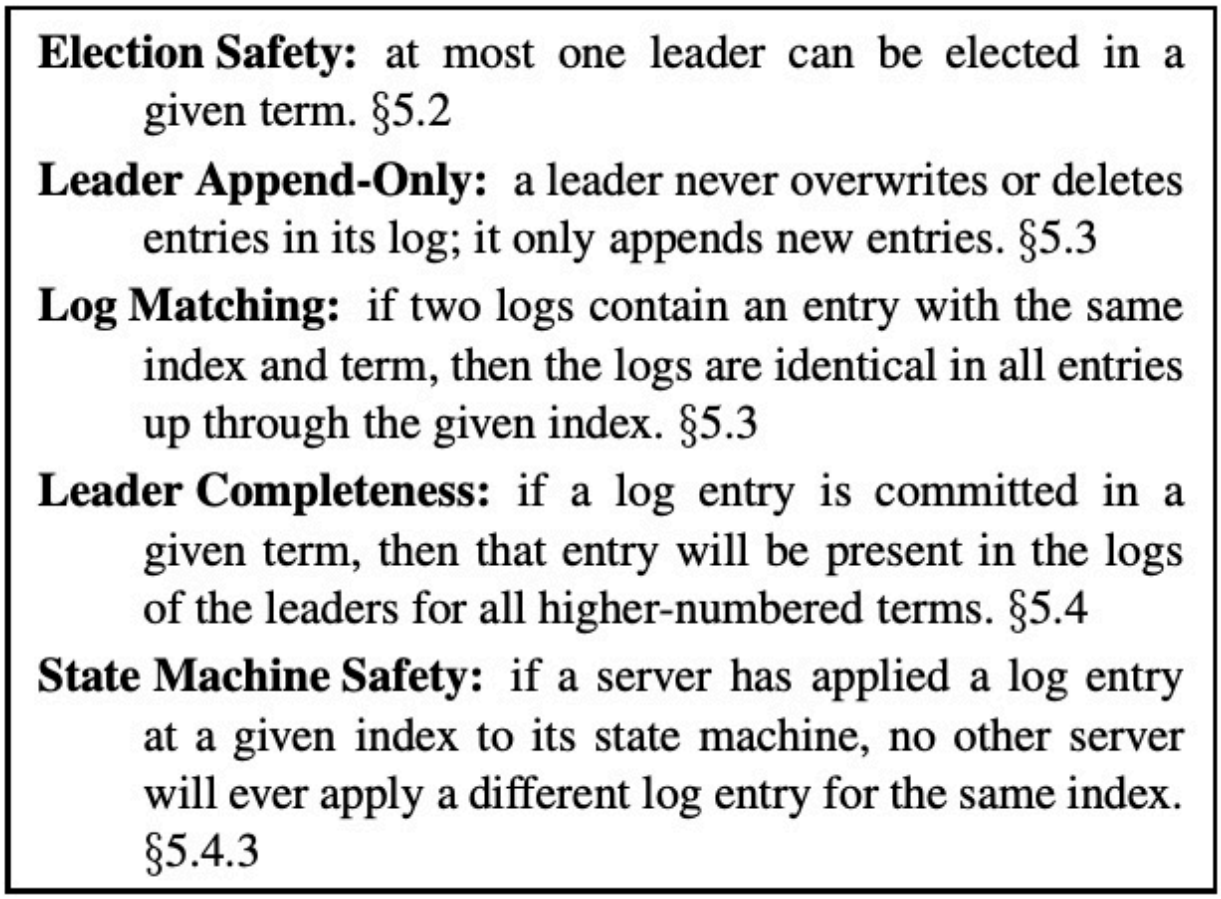
The Raft consensus algorithm

Raft works first and foremost with leaders. Leaders are elected, they receive client calls, they handle requests to other Raft nodes, they decide what the replicated log should look like. If a leader fails or its term expires, a new leader is elected.

Zooming out, Raft has three subproblems to focus on:

- **Leader election** - "a new leader must be chosen 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 ... if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index."

And Raft makes the following guarantees at all times:



5.1 - Basics

A node in a Raft cluster is always in exactly one of three states:

1. **Leader** - manages consensus and all followers; handles all client requests
2. **Follower** - passive, receives information from leaders
3. **Candidate** - used for new leader election

Raft nodes have a shared sense of time via **terms**. Terms are arbitrary slices of time labelled by monotonically increasing integers. Each term begins with an election for a leader.

Terms allow for a shared sense of time. The current / latest term for a node is shared in every communication between nodes. Stale terms revert leaders to followers / candidates. Requests with stale terms attached are rejected. Remember, nodes might miss whole terms - they need to know what everyone else thinks!

Last basic - raft uses **RPC calls** for all operations. Just two:

1. **AppendEntries**: used by the leader to distribute log state
2. **RequestVote**: issued by candidates during elections.

All RPC calls have retry semantics and are issued in parallel.

5.2 - Leader Election

Leaders heartbeat to maintain their authority. Followers are cool with that, and will only start an election if they don't receive heartbeats (empty `AppendEntries` RPCs) within the *election timeout* period.

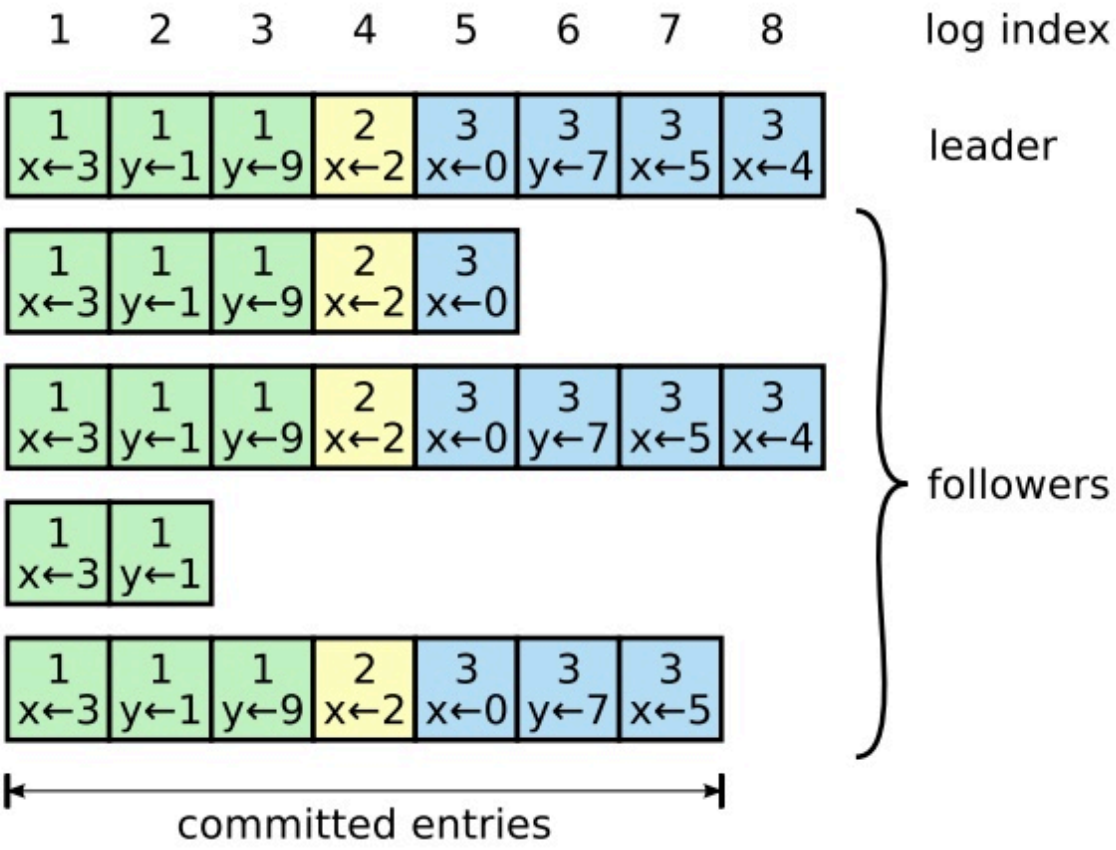
Once a follower starts an election, they become a candidate, increment their term, and issue `RequestVote` RPCs in parallel to all other nodes. If they win the election (votes from nodes are first-come, first-serve, they become the leader by sending heartbeats to all other nodes (to prevent any other elections). Similarly, they'll go back to follower if they get a leader heartbeat with a term greater than or equal to its own term.

If lots of election timeouts happen at once, split votes can occur. Raft randomizes election timeouts (e.g. varies between 150 - 300 ms), which prevents split votes. Election timeouts are observed at the beginning of each node's election attempt (note - randomized election timeouts was chosen as the split vote breaker for its simplicity!).

5.3 - Log Replication

A leader services client requests by appending the command to its own log, then issuing corresponding `AppendEntries` RPC calls in parallel to nodes. When the entries are safely replicated, it applies the log to its state machine.

Logs look like this:



Log entries are **committed** once they've been persisted to a majority of node logs. Anything committed will eventually be executed on each node's state machine. The leader knows the highest-committed log index, and sends that info to followers. In this manner, all preceding log entries in addition to the latest will be sent as part of commit stage - more on that to come.

When an `AppendEntries` RPC is issued, followers will perform a simple consistency check. The RPC will include the index and term of the log immediately preceding the newest entries -- if the follower does not have an identical term and index at its tail, it rejects the RPC.

This allows for eventual consistency among Raft nodes. The leader will discover the point in the log at which the leader and follower diverge, delete all subsequent entries on the follower, and replicate its own leader entries. In this manner, the leader maintains consistency by force replicating its own logs.

The leader maintains a `nextIndex` per follower, which is initialized to `maxLeaderIndex + 1` upon election. If a leader and follower disagree on term and index in the consistency check in `AppendEntries`, the `nextIndex` will decrement until they agree. Then all of those leader log entries at the point of initial divergence will be replicated.

5.4 - Safety

This isn't quite enough. We need to make sure any elected leader contains all committed log entries from prior terms.

During the election process, a candidate will figure out if it's at least as up-to-date as all other nodes voting for it. If it is, e.g. its latest term and index is greater than or equal to all other nodes, it can be leader. Otherwise, no good.

Raft will also never commit entries from prior terms, even if that entry appears on a majority of nodes. It will simply replicate its log via the `AppendEntries` + consistency check mechanisms above, and then all nodes will be up-to-date.

The safety proof is a proof by contradiction. Say we have the following bad situation: a leader with a committed entry, and a future leader that doesn't contain that entry. This leads to a contradiction where one node must have both voted for the new leader and received the replicated entry from the first leader. This is impossible based on the given consistency check on `AppendEntries`.

5.5 - Follower and candidate crashes

If a follower or candidate fails, requests sent to it fail. Those requests are retried indefinitely. If a request is completed by a follower, but not responded to, that request will be retried. Raft RPC requests are idempotent, so this is fine.

5.6 - Timing and availability

Timing must be correct so that Raft can make progress. This will happen as long as your RPC time is much less than your election timeout, which must be much less than your average node failure time.

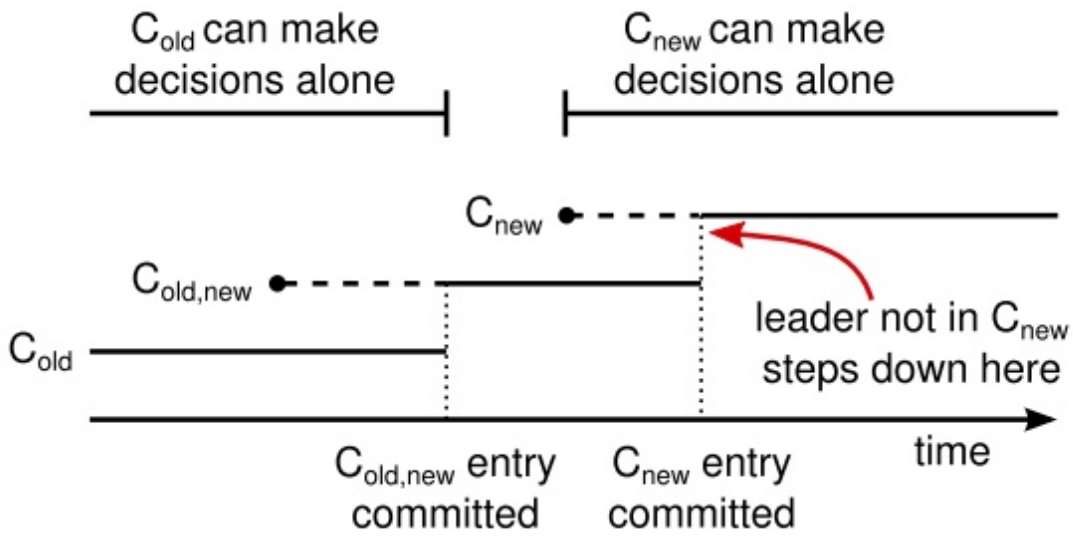
> broadcastTime ≪ electionTimeout ≪ MTBF

6 - Cluster membership changes

Cluster membership can change over time. It's not feasible to do these changes atomically, so instead Raft opts for a two-phase operation called `joint consensus`. This means that the cluster observes a brief transition period in which servers from both old and new configuration are involved in the consensus process. More specifically:

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

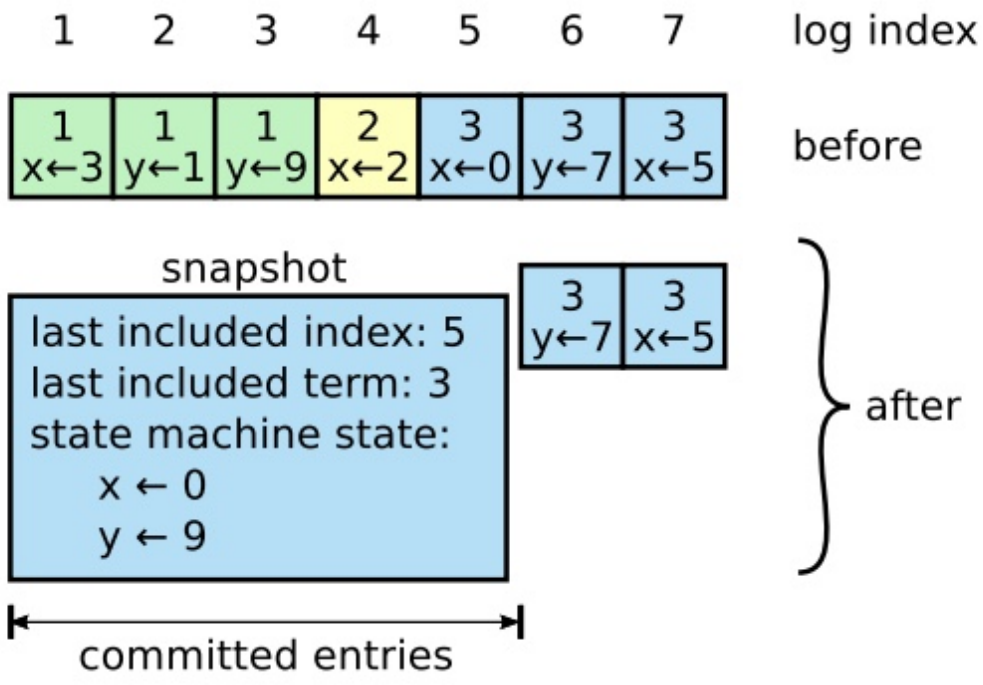
The config is applied to the leader, which stores the joint config `C(old,new)` - this config is replicated to and used by all servers. Once `C(old,new)` is committed, the leader can starting replicating `C(new)` to be used. This will eventually be committed. As the following image portrays, there is no time in which both the old and new configurations can unilaterally make decisions:



A couple of pending issues with this setup:

1. New nodes might not have any log entries and could take awhile to replicate the old leader's log. Raft uses a new phase right before the config changes to allow new nodes to stand as non-voting members of the cluster - e.g., they have no say in consensus, but leaders will replicate log entries to them.
2. An old node may end up managing the new cluster briefly - the old cluster leader will step down once `C(new)` is fully committed. While it's managing the new cluster, it won't count itself in majorities.
3. Removed servers cause issues with heartbeating and timeout, getting stuck in cycles of elections. `RequestVote` RPCs can be ignored when a follower thinks a leader exists in this case. I don't fully follow this one.

7 - Log Compaction



Snapshots are good for log compaction. Aggregate the state, keep the latest config with it, and then use that as

your base for the rest of the log.

There's a new RPC, `InstallSnapshot`, which can be used by leaders to send its snapshot to followers that have fallen sufficiently far behind that the leader doesn't actually have their individual log entries any more (only the snapshot).

Followers can take their own snapshots. This is different from the rest of Raft's *"strong leader"* principle, but it's justified for the sake of understanding.

Beyond that, nodes just have to figure out how often to snapshot, and how to do it performantly. Performance can be via copy-on-write semantics (can still serve requests while doing the snapshot). Frequency is probably more individualized.

8 - Client interaction

Raft provides linearizable semantics to clients. Clients will connect first to any node, and then forwarded on to the current term leader.

Clients might retry a committed but unresponded request - this would break the linearizable semantics. To avoid that, each leader stores a unique id for the last request it processed per client, along with its response. If it gets a duplicate, it'll just serve that response immediately. In this manner, idempotency is used to preserve linearizability.

To preserve linearizability on read-only requests, leaders cannot return stale data. To avoid this, Raft employs two mechanisms:

1. Leaders need to know that they have all the latest entries. "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". Raft leaders do a blank no-op log entry at the start of each term to avoid this. I don't fully get this.
2. A leader needs to know if it's been deposed. It does this by heartbeating with all nodes on each read request.

10 - Related work

Lots of Paxos (Spanner + Chubby). Zab via ZooKeeper. Viewstamped Replication. TiKV nowadays.

Ultimately, the biggest difference between Raft and Paxos is Raft's use of strong leaders. This simplifies reasoning a LOT.