Raft

> Implementation notes, to refresh my memory and make code-writing easier.

Raft is an algorithm for managing a replicated log. It breaks consensus into a number of simple sub-problems, namely:

- leader election
- log replication
- safety

The guiding principle behind its design is *understandability*. We are taking a log and replicating it over a set of state machines.

Algorithm Basics

A cluster contains a relatively small number of servers – typically 5. Each can be in one of three states:

- leader
- follower
- candidate

During normal operation there's one leader and all the others are followers. Leaders assume responsibility for the log, handling client requests. Followers are passive, only responding to messages sent their way. Candidates try to become leaders when they think there isn't one (due to the leader failing, for example).

Raft's concept of time is *terms*, which are numbered and increase monotonically. Terms begin with an *election*, in which one or more candidates try to become the leader. A candidate that wins an election becomes a leader for the rest of the term. If the vote is split, a new election begins.

Each server keeps track of the current term number, and exchanges this number as part of each communication. This makes it easy to discover when a node has become out of date, in which case it becomes a follower (if necessary). Also, stale requests from lower term numbers can easily be ignored.

There are two RPCs:

- RequestVote: used by candidates to solicit votes during elections
- AppendEntries: used by leaders to replicate log entries (and as heartbeats)

Note, there is an additional `InstallSnapshot` RCP for configuration changes.

Leader Election

Nodes boot up as followers, and remain followers as long they are receiving RPCs from a leader (heartbeats and AppendEntries) or candidate (RequestVote). If no communication is received over an election timeout, the node assumes that there isn't a leader and begins an election.

This is done by incrementing `self.term`, and transitioning to a candidate state by voting for itself and sending out `RequestVote` RPCs to all peers. This state is maintained until either:

- it wins an election
- another server wins an election
- nobody wins

The node wins by receiving votes from a majority of servers in the full cluster for that term. Once it has this, it sends out a set of heartbeat messages to all peers to establish its authority for that term (thus preventing additional elections).

If while waiting, the node receives an `AppendEntries` RPC from another node, it checks if the term in the RPC is at least as large as its current term. If it is, it recognizes this as the leader and becomes a follower. Otherwise, it rejects the RPC and continues.

If the vote split, then the candidate will end up timing out. Thus, a new election begins. Timeouts are randomized to minimize the probability of persistent collisions.

Log replication

Given a client request, the leader appends to its log and then issues `AppendEntries` RPC to every peer. Once the log entry has been replicated by the majority of machines, it is committed (along with all previous entries in the leader's log). This means that the leader applies the entry to its own state machine and returns a success to the client, guaranteeing that it'll eventually be executed by all available nodes. The leader keeps track of the highest committed entry and communicates these in `AppendEntries` RPCs so other servers can find out and apply them.

The *Log Matching Property* ensures the following properties:

- two entries in different logs with the same term and index store the same command
- the entire log preceding two entries in different logs with the same index in term are identical

In the case of inconsistencies, the leader forces followers to duplicate its own log. To do this, the leader finds the last entry on which they agree, and has the

follower overwrite everything after that with entries from its log. Note that the leader never overwrites of deletes anything from their own log.

Safety

To become leader in a given term, a node must have all of the entries committed in previous terms. This is discovered in the election process – because a candidate must get votes from a majority of nodes, there must be some overlap that replicated the last committed entry.

Follower and candidate failures

Handling follower and candidate failures are simple – message sent to these won't arrive, and will be retried indefinitely. This works out okay because all RPCs are idempotent.