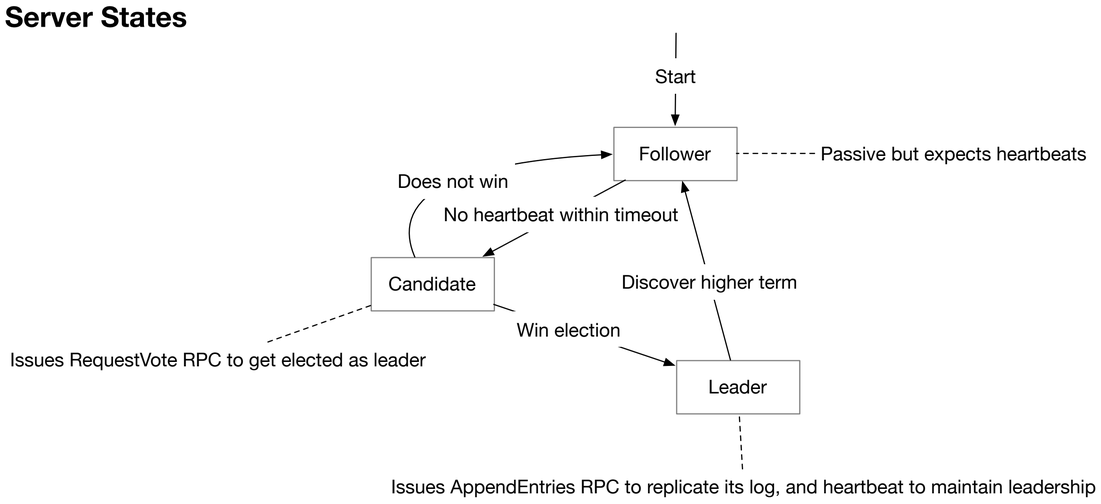
Server States



Two types of RPC:
1. **RequestVote**
2. **AppendEntries** (append an entry or serve as heartbeat)

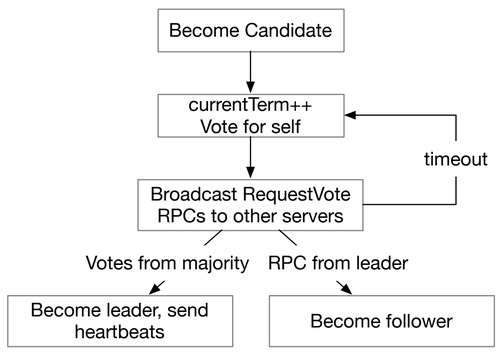
How do I know I won / lost an election?
How is liveness guaranteed?

Each server maintains current **term** value. Rationale: need to be able to identify obsolete information
1. At most 1 leader per team
2. Some terms have no leader (failed election)
3. Term is exchanged in every RPC
4. If peer has later term, update term and revert to follower
5. If incoming RPC has obsolete term, reply with error

There is no global view of term. Each server keeps its own understanding of what the current term is, and exchange uses the above mechanism.
Over time term converges to the latest value.
The only concept of time elapse in the system, is the timeout, and then each one's perception of term.

How does one decide to start a new term?
What would happen without a term?

Leader Election



What happens if a server receives a RequestVote, votes for it, and then receive another RequestVote? Does it just ignore the later request?
Would term values in these two requests matter?
How would I know I get majority? (Cluster membership management)

Safety: at most one winner per term. Guaranteed by:
1. at most one vote per server per term. This vote is persisted on disk, such that if it crashes before a leader can be elected, a vote won't be recast.
2. Majority is required to win election

Liveness: some candidate must eventually win
1. Choose election timeouts randomly in [T, 2T]. (Random numbers are your friend! Distributed systems need random numbers)
2. One server usually times out and wins election before others time out
3. Works well if T >> broadcast time

While this can work well in practice (when it's easy to set T >> broadcast time), what happens if T is almost the same as broadcast time, or 2T is smaller than broadcast time?

Alternatively to randomization, ranking (like biggest proposal numbers win in Paxos) can bring liveness as well.
Using randomization over ranking demonstrates design for simplicity.

Statement: leader election is kind of like the first phase of Paxos, log replication is kind of like second phase of Paxos. How to interpret this?

Normal Operation

Client sends command to leader.
Leader appends command to its log.
Leader sends AppendEntries RPCs to all followers.
Once leader decides an entry is **committed**
  1. Leader executes command in its state machine, returns result to client
  2. Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  3. Followers execute committed in their state machines

Committed is defined as "this command is replicated on a majority of servers by leader of its term", a command is committed means that it's safe to execute.
  (*Thus each log entry stores the term*?, and a command, like x = 5; does *by leader of its term* hold any significance?)
Each log entry contains the term value, its index in log history, and the command.

If a follower crashes or is slow, the leader keeps retrying until they succeed

Safety (derived from log matching property and leader completeness):
Log can go into an inconsistent state, for example, a leader receives many commands, before it can replicate all of them, it crashes.
A new server becomes leader, and it doesn't know the commands the previously crashed server received.
To simplify recovery from log inconsistency, **leader assumes its log is perfect**.
Normal operation will repair all inconsistencies, to maintain **log matching property**: goal being high level of consistency between logs.
Log matching property dictates
1. If log entries on different servers have the same index and term, then they store the same command, and the logs are completely identical up to this point.
2. If one entry is committed, all preceding entries are committed

To replicate a log entry from leader to a follower, entries before it have to be reconciled.
To do this, in AppendEntries RPC, to replicate a new entry, leader includes the index and term value of its preceding entry.
Followers must contain preceding entry, otherwise it rejects leader's RPC, and leader retries with lower index (include one more preceding entry in its request), until it finds a match with the follower, and the leader is able to rewrite follower's log after the matched point.
This rule guarantees that log entries are only appended if the follower is in sync with the leader.

Leader completeness: how can we make sure leader's log is perfect?
Once log entry is committed, all future leaders must store that entry. Servers with incomplete logs must not get elected.
This can be done as candidates include index and term of its last log entry in RequestVote RPCs, and voting server denies vote if its own log is more up to date (as defined by first comparing term, if mine is bigger mine is more complete, if equal terms but my last index is larger, mine is more complete).
With this added rule to election, a server can only be elected if it contains logs that are more complete than half of the voting servers; meaning, only those that have stored the last committed entry (by definition, replicated to a majority, use example to illustrate!) have a chance of being elected.

The whole cluster can make progress (elect leaders, replicate logs) as long as majority is up
In the face of a network partition, as long as majority is maintained, the algorithm still works