properties

* **Election safety**: at most one leader can be elected in a term
* **Leader Append-Only**: leader never overwrites or deletes entries in its log
* Log Matching: if two logs contain an entry with same index and term, then logs are identical up to that index
* **Leader Completeness**: if log entry committed in given term, that entry present in the logs of the leaders for all higher-numbered terms
* **State Machine Safety**: if 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 same index

Leader election

* server remains follower as long as it receives valid RPCs from leader or candidate
* Leaders send periodic heartbeats to all followers
* If no heartbeat within election timeout, follower begins election
  + increment current term and transition to candidate state
  + vote for itself and issue RequestVote RPC to other servers
* Each server votes for at most 1 candidate in a term
  + first come first serve
* Candidate wins if majority of servers vote for it
  + Once wins, issues heartbeat messages
* If a candidate receives AppendEntries from another server and that server has term at least as large as candidate, candidate recognizes that server as leader
* If split vote, use randomized election timeout to start again

Log Replication

* When client request command to be executed
  + leader appends command to log
  + AppendEntry RPCs to other servers to replicate entry
* Committed
  + means entry is safe to apply to state machine
  + entry is committed once leader that created entry replicated it on majority of servers
    - Also commits all preceding entries in leader's log
  + Leader keeps track of highest index it knows to be committed
    - sends this to the followers
    - Followers can apply them
  + If two entries in different logs have same index and term, then they store same command
    - leader creates at most one entry with given log index in a given term, and log entries never change position in log
  + **Log Matching**
    - from consistency check in AppendEntries
    - in the RPC, leader includes index and term of entry in log right before nextIndex
      * if follower doesn't match it, refuses new entries
  + NextIndex
    - when leader first comes to power, nextIndex is set to last index in log plus 1
    - If failed AppendEntries, leader decrements nextIndex
      * follower sends the term of the entry it has at prevLogIndex
      * follower sends the earliest entry it has for that term
      * leader tries to find that term in its own log, and sets nextIndex to be the last entry it has of that term
  + Leader never overwrites or deletes entries in its own log
  + **Leader Completeness Property**
    - If candidate elected, will contain all committed entries
    - proof
      * supposed for contradiction does not hold
      * LeaderT for term T commits entry from its term, but log entry not stored by leader of some future term
      * Consider smallest term U > T whose LeaderU does not store entry
      * LeaderT replicated entry on majority of clusters and majority voted for LeaderU
      * So at least one server who did both
      * the server must have accepted committed entry from T before voting for LeaderU
        + otherwise would have rejected AppendEntries from LeaderT
      * server still had the entry when voted for LeaderU since assume every leader before that had the entry
      * LeaderU log should be at least as up to date as voter
        + **up-to-date** = last log term higher or same but longer log
      * If the two servers shared same last log term, then leaderU log must be at least as long as voter's, so log contained every entry of voter's log. Contradiction!
      * Otherwise leaderU's last log term must have been larger than voter's
        + The earlier leader that created leaderU's log entry assumed to have contained committed T entry, so by Log Matching, leaderU's log must also contain committed entry. Contradiction!
  + Leader can only be sure of committed entries on its term
    - once entry in its own term has been committed, all prior entries committed
  + Timing
    - Raft able to elect and maintain steady leader with timing requirement:
      * broadcastTime << electionTimeout << MTBF
      * << = way less than
      * broadcastTime = average time takes server to send RPCs in parallel to every server and receive responses
      * MTBF is average time between failures for a single server
        + When leader crashes, system unavailable for approximately election timeout time, so should be able to elect new leader before it also crashes
  + Snapshotting
    - Snapshot to prevent log from being too long
    - Snapshot only committed entries
    - save last included index and term in snapshot
    - leader can send snapshots to followers
      * when nextIndex is in the leader's snapshot
    - InstallSnapshot RPC
      * if snapshot contains new info, follower discards entire log
      * If snapshot describes prefix of follower's log, log entries covered by snapshot are deleted but entries following snapshot are still valid
  + Clients
    - Clients send all requests to leader
    - If server is not leader, server will tell client which one is leader
    - Clients will choose random server if leader not responsive
    - Clients can assign unique serial numbers to every command so do not do a command twice
    - Linearizable reads cannot return stale data
      * leader must have latest info on which entries committed
      * Leader commits blank no-op entry into log at the start of term to see which entries committed before term began
      * Leader must also check if still the leader before processing read-only request
      * So leader exchanges heartbeat messages with majority of cluster before responding to new requests