**Raft**

**Introduction**

* Raft is a consensus algorithm for managing a replicated log
* Consensus algorithms allow collection of machines to work as group to survive failures of some of its members
* Novel features:
  + **Strong Leader**: Log entries only flow from leader to other servers
  + **Leader Election**: Randomized timers to elect leaders adding only small amount of overhead to heartbeats which are already required
  + **Membership Changes**: Joint consensus approach when switching set of servers in cluster in which majorities of the servers overlap during transitions

**Replicated State Machines**

* Replicated state machines compute identical copies of the same state and continue doing so even if some servers go down
* Large-scale systems with a single cluster leader typically use a separate replicated state machine to manage leader election and store configuration information that should survive if leader crashes
* Replicated log—each server stores log with series of commands that state machine executes in order
* Logs have same commands in same order so each state machine processes same sequence of commands
* State machines are deterministic therefore each computes same state and outputs
* The consensus algorithm keeps the replicated log consistent
* Consensus module on a server receives commands from clients and adds them to its log
* It also communicates with consensus modules on other servers to make sure every log contains same requests in same order even if some servers fail
* Once commands are properly replicated, each server’s state machine processes them in the order they appear in the log and outputs are returned to clients
* Consensus algorithm properties:
  + **Safety**: Never return an incorrect result under all non-Byzantine conditions including network delays, packet loss, duplication, reordering, etc.
  + **Availability**: Fully functional as long as majority of servers are operational and can communicate with each other and clients
  + **Time Independent**: Do not depend on timing to ensure consistency of logs; faulty clocks and extreme message delays cause availability problems at worst
  + **Responsiveness**: Common case is when command completes as soon as majority of cluster has responded to single round of RPCs; minority of servers need not slow down performance

**Raft Consensus Algorithm**

* Leader is first elected—complete responsibility over managing replicated log
* Consensus subproblems:
  + **Leader Election**: New leader must be chosen when existing leader fails
  + **Log Replication**: Leader must accept log entries from clients and replicates them across cluster, forcing other logs to agree with its own
  + **Safety**: If any server has applied particular log entry to its state machine, then no other server may apply a different command for the same log index

**Raft Basics**

* Each server is either leader, follower, or candidate
* Common case is with one leader and all the others are followers
* Raft divides time into terms of arbitrary length
* Each term begins with an election in which one or more candidates tries to become leader
* If a candidate wins, then it serves as leader for the rest of the term
* In a split vote, the term ends with no leader and new term begins shortly
* There is at most one leader per term

Diagram

Description automatically generated

* Different servers observe transitions between terms at different times and sometimes a server may not observe an election or even entire terms
* Terms are exchanged whenever servers communicate; if one server’s term is less than another server’s term it is updated to the larger value
* If a candidate or leader finds its term is out of date, it immediately reverts to a follower state
* Raft servers communicate with RPCs
  + RequestVote RPCs are initiated by candidates during elections
  + AppendEntries RPCs are initiated by leader to replicate log entries and provide a form of heartbeat

**Leader Election**

* Heartbeat mechanism to trigger leader election
* Servers start as followers and remain in follower state as long as they receive valid RPCs from a leader or candidate
* Leaders send periodic heartbeats (AppendEntries RPCs carrying no log entries)
* If a follower receives no communication over a period of time (election timeout) then it assumes there is no leader and begins election
* To begin election
  + Follower increments term
  + Transitions to candidate state
  + Votes for itself
  + Issues RequestVote RPCs in parallel to each other server
  + Candidate continues until one of three things happens
    - Wins election (majority votes from servers)
    - Another server is leader
    - Period of time goes without winner
* Outcome 1: Once a candidate wins an election, it sends heartbeat messages to all of other servers to establish authority and prevent new elections
* Outcome 2: While waiting for votes, candidate may receive AppendEntries RPC from another server claiming to be leader
  + If leader’s term is at least as large as candidate’s current term, candidate recognizes leader as legitimate and goes to follower state
  + If leader’s term is smaller than candidate’s current term, the candidate rejects RPC and continues in candidate state
* Outcome 3: Candidate neither wins nor loses election
  + Each candidate will time out and start new election by incrementing term and initiating another round of RequestVote RPCs
* Randomized election timeouts ensure split votes are rare
* To prevent split votes, election timeouts are chosen randomly from a fixed interval which ensures that in most cases only a single server will time out, win the election, and send heartbeats before any other servers time out
* To handle split votes, each candidate restarts its randomized election timeout at the start of an election and waits for that timeout to elapse before starting the next election