|  |
| --- |
| In Search of an Understandable Consensus Algorithm |
| Raft is a consensus algorithm for managing a replicated log.  It produces a result equivalent to (multi-)Paxos |
| replicated state machines：  a collection of servers computing identical copies of the same state |
| Paxos cons:   1. difficult to understand. 2. not provide a good foundation for building practical implementations (single-decree) 3. a symmetric peer-to-peer approach (not simple and fast than having a leader) |
| Raft:  separated leader election, log replication, safety, and membership changes |
|  |
| properties: |
| five servers to tolerate two failures |
| server states in one of the three: leader, follower, or candidate |
| In normal operation there is exactly one leader and all of the other servers are followers. |
| Followers are passive: simply respond to requests from leaders and candidates. |
| The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader) |
| candidate, is used to elect a new leader |
|  |
|  |
| Raft ensures that there is at most one leader in a given term. |
| Terms act as a logical clock in Raft, and they allow servers to detect obsolete information |
| Term exchange:  Each server stores a current term number, which increases monotonically over time.  Current terms are exchanged whenever servers communicate   1. if one server’s current term is smaller than the other’s, then it updates its current term to the larger value 2. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. 3. If a server receives a request with a stale term number, it rejects the request |
| RequestVote RPCs are initiated by candidates during elections  Append-Entries RPCs are initiated by leaders:   1. replicate log entries 2. heartbeat   Servers retry RPCs if they do not receive a response in a timely manner |
|  |
| Leader election |
| To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. Until:  (a) it wins the election,( wins an election if it receives votes from a majority of the servers in the full cluster for the same term.)  (b) another server establishes itself as leader,  (c) a period of time goes by with no winner (time out [randomized election timeouts] and start a new election by incrementing its term and initiating another round of Request-Vote RPCs.) |
| Each server will vote for at most one candidate in a given term, on a first-come-first-served basis. |
| Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections. |
| a candidate may receive an AppendEntries RPC:   1. If the leader’s term is at least as large as the candidate’s current term, then the candidate recognizes the leader as legitimate and returns to follower state. 2. Otherwise, reject and remain candidate |
| election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms) |
| The same mechanism is used to handle split votes |
|  |
| Log replication |
| Procedure:   1. Each client request contains a command 2. The leader appends the command to its log as a new entry 3. the leader issues AppendEntries RPCs to other servers (retries Append-Entries RPCs indefinitely until all followers eventually store all log entries) 4. the leader applies the entry to its state machine and returns the results to the client |
| Log structure: |
| Each log entry stores a state machine command along with the term number  Each log entry also has an integer index identifying its position in the log |
| Log commit:   1. The leader decides when it is safe to apply a log entry to the state machines 2. log entry is committed once the leader that created the entry has replicated it on a majority of the servers   This also commits all preceding entries in the leader’s log, including entries created by previous leaders   1. The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs 2. Once a follower learns that a log entry is committed, it applies the entry to its local state machine |
| When receiving an AppendEntries RPC, If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. |
| When inconsistent:   1. leader crashes can leave the logs inconsistent 2. conflicting entries in follower logs will be overwritten with entries from the leader’s log |
| Maintain consistency:   1. find the latest log entry where the two logs agree, 2. delete any entries in the follower’s log after that point 3. send the follower all of the leader’s entries after that point.   All of these actions happen in response to the consistency check performed by AppendEntries RPCs.  Details:   1. The leader maintains a nextIndex for each follower, which is the index of the next log entry the leader will send to that follower 2. it initializes all nextIndex values to the index just after the last one in its log 3. After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC.   With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. |
| A leader never overwrites or deletes entries in its own log |
| in the normal case a new entry can be replicated with a single round of RPCs |
|  |
| Safety |
| ensures that the leader for any given term contains all of the entries committed in previous terms |
| log entries only flow in one direction, from leaders to followers |
| RequestVote RPC includes information about the candidate’s log, and the voter denies its vote if its own log is more up-to-date than that of the candidate |
| If the logs have last entries with different terms, then the log with the later term is more up-to-date.  If the logs end with the same term, then whichever log is longer is more up-to-date |
| Raft never commits log entries from previous terms by counting replicas. |
| log entries retain their original term numbers when a leader replicates entries from previous  terms. |
| Follower and candidate crashes:  Raft handles these failures by retrying indefinitely  Raft RPCs are idempotent |
|  |
| timing requirement:  broadcastTime≪electionTimeout≪MTBF |
| Raft’s RPCs typically require the recipient to persist information to stable storage, so the broadcast time may range from 0.5ms to 20ms  As a result, the election timeout is likely to be somewhere between 10ms and 500ms. |
|  |
| Cluster membership changes |
| the cluster can potentially split into two independent majorities during the transition |
|  |
|  |
|  |
|  |