Raft

Miguel Alves, Nº 49828

1 Introduction

The **Raft** protocol is used in the context of log-replication, ensuring consensus on the order in which the log entries will be applied.

Each entity involved in the protocol is called a **peer**, and in any given instant a peer can have one of 3 roles - **Follower**, **Candidate** and **Leader**.

2 Internal State

The following is the state stored by each peer. To improve readability, the state is split amongst the different roles - Follower, Candidate and Leader - that each peer can have.

2.1 Common State

The following is the state stored by each peer, regardless of what role it is in:

1. `pid` → the peer's identifier
2. `current_term` → the latest term that this peer has seen
3. `voted_for` → the `pid` of the candidate who this peer voted for in the previous election, or `null`
4. `current_leader` → the `pid` of the peer believed to be the leader
5. `log` → a list of log entries, where each entry is a pair < term, operation >
6. `role` → this peer's role
   (a) `CANDIDATE` → has begun an election and wants to become the leader
   (b) `LEADER` → is the leader
   (c) `FOLLOWER` → is a follower
7. `commit_index` → the index of the highest log entry known to be commited
8. `last_applied` → the index of the highest log entry whose operation this peer has applied
9. `membership` → the list of known correct peers
10. `election_timeout` → the time when the next election timeout will occur if the peer does not receive a message from a valid Leader

2.2 Candidate State

The following state is stored by peers when their role is **Candidate**:

1. `votes_rcvd` → the list of votes this candidate has received from other peers

2.3 Leader State

The following state is stored by peers when their role is **Leader**:

1. `next_index` → a list where, for each peer, stores the next log entry to send to that peer
2. `match_index` → a list where, for each peer, stores the index of the highest log entry known to be replicated on that peer
3. `heartbeat_timeout` → the time when the Leader should send an append entries request, even if there are no new entries, to inform the peers that it hasn't failed

3 Message Structure

The following is the base structure of the messages sent between peers in the **Raft** protocol. Note that implementing the protocol for different programming languages and use cases may require adding more information to these messages.

- `RequestVote`
  1. `candidate_term` → the `current_term` of the candidate that is requesting the vote
  2. `candidate_pid` → the `pid` of the candidate that is requesting the vote
  3. `last_log_index` → the index of the candidate's last log entry
  4. `last_log_term` → the term associated with the candidate's last log entry
- `ResponseVote`
  1. `follower_term` → the `current_term` of the follower that is responding to the vote
  2. `follower_pid` → the `pid` of the follower that is responding to the vote
  3. `vote_granted` → the follower's response to the vote. `TRUE` if the follower voted for the candidate, `FALSE` otherwise
- `RequestAppend`
  1. `leader_term` → the `current_term` of the leader that is requesting the append entries operation
  2. `leader_pid` → the `pid` of the leader
  3. `prev_log_index` → the index of the log entry that immediately precedes the new entries being sent
  4. `prev_log_term` → the term associated with the entry at `prev_log_index`
  5. `entries` → a list with the new entries being sent (or empty, if it is an heartbeat)
  6. `leader_commit_index` → the `commit_index` of the leader
- `ResponseAppend`
  1. `leader_pid` → the `pid` of the leader that initially request the append
  2. `follower_term` → the `current_term` of the follower that is responding to the append
  3. `follower_pid` → the `pid` of the follower that is responding to the append
  4. `success` → `TRUE` if the follower appended the entries to it's log, `FALSE` otherwise
  5. `match_index` → the index of the last log entry that the follower has in it's log
- `RequestOperation`
  1. `operation` → the operation to be added to the log

4 High Level Description

4.1 Follower

When a peer has the **Follower** role, it's behavior is based on receiving and responding to messages.

Upon receiving a `RequestOperation` message from a client, the Follower redirects that message to the current leader.

Upon receiving a `RequestVote` message from a Candidate, the Follower responds with a `Response-Vote` message. The vote will be `TRUE` is the Candidate's term and log entries are up-to-date, and the Follower hasn't in favor of another Candidate. Otherwise, the vote with be `FALSE`.

Upon receiving a `RequestAppend` message from a Leader, the Follower will respond with a `Response-Append` message. The message will contain `success=TRUE` if the entries have been appended to the Follower's log or `success=FALSE` if a conflict was detected - the Follower is missing logs, or the Leader is outdated (the Leader's term is smaller than the Follower's term).

If the `RequestAppend` succeeds, the Follower will also check the Leader's `commit_index` (included in the request) and will update it's own `commit_index`, applying the necessary operations.

A Follower will change it's role to **Candidate** when a heartbeat_timeout occurs - more than X amount of time has elapsed since the Follower received a message from a valid Leader.

4.2 Candidate

When a peer changes it's role to **Candidate**, the election process begins. The Candidate starts by incrementing it's `current_term` and sending a `RequestVote` message to all the other peers.

Upon receiving a negative `ResponseVote` message, the Candidate will check if the `follower_term` included in the message is larger than it's `current_term` and if that is true, the Candidate will revert it's role back to Follower since there is another peer that is more up-to-date.

Upon receiving a majority of positive `ResponseVote` messages, the Candidate will change it's role to **Leader**.

4.3 Leader

The **Leader** is in charge of sending `RequestAppend` messages to all the other peers every **X** amount of time, regardless of whether or not there are new entries in it's log, because these messages also serve the purpose of informing the peers that the Leader did not fail.

Upon receiving an unsuccessful `ResponseAppend` message, the Leader will decrement the `next_-index` of the corresponding peer because there was a conflict. Worst case scenario, the conflict is in the start of the log and therefore the request will only succeed when the Leader decrements the `next_index` to 0.

Every time the Leader receives a successful `ResponseAppend` message, it will check what is the latest entry that it can `commit`, which will be the entry with largest index that is replicated in a majority of peers.

Upon receiving any message that contains a `term` larger that it's own, the Leader will revert it's role to **Follower** since that means there is another Leader that is more up-to-date.

5 Role Changes

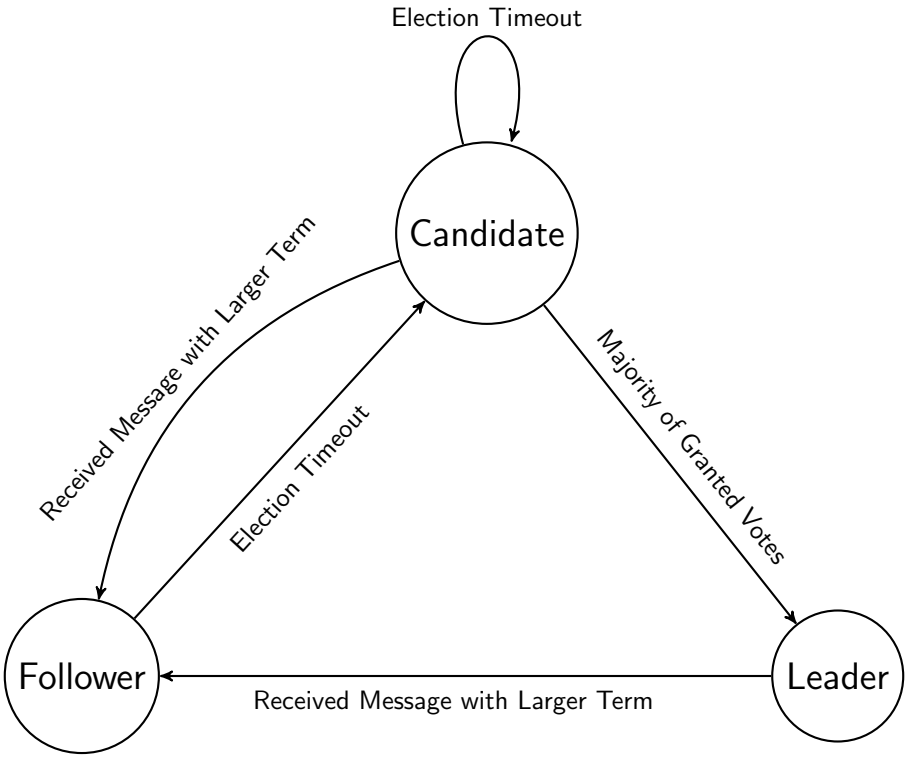

Figure 1: State Machine Diagram that shows how a peer transitions between different roles.

6 Internal State Changes

The following are the steps where there are internal state changes.

1. Sending entries to peers (`send_entries`)
   (a) `pre_send_entries`
       assert(role == LEADER)
   (b) `post_send_entries`
       heartbeat_timeout = current_time + HEARTBEAT_TIMEOUT
2. Starting an election (`begin_election`)

   current_term += 1
   role = CANDIDATE
   voted_for = self
   votes_rcvd = [ ]
   election_timeout = current_time + ELECTION_TIMEOUT
3. Sending a RequestVote to a peer (`request_vote`)
   (a) `pre_request_vote`
       assert(role == CANDIDATE)
       assert(peer **not in** votes_rcvd)
4. Sending a RequestAppend to a peer (`append_entries`)
   (a) `pre_append_entries`
       assert(role == LEADER)
5. Becoming a Leader (`become_leader`)
   (a) `pre_become_leader`
       assert(role == CANDIDATE)
       assert(length(votes_rcvd) > (length(membership) / 2))
   (b) `post_become_leader`
       role = LEADER
       current_leader = self
       next_index = list of size **length**(membership) with each index initialized as **length**(log)
       match_index = list of size **length**(membership) with each index initialized as 0
6. Updating the commit index (`update_commit_index`)
   (a) `pre_update_commit_index`
       assert(role == LEADER)

   for index **in** commit_index+1 **to length**(log)
       var replicas = 0
       for peer_match **in** match_index
           if peer_match ≥ index **then** replicas++
       if replicas > **length**(membership)/2 **then** commit_index += 1

7. Receiving a request from a client (`receive_client_request`)

   (a) `case_leader`

   log.add(Entry<operation, current_term>)

   (b) `case_non_leader`

   — redirect to leader —

8. Receiving a valid RequestVote from a candidate (`handle_vote_request`)

   (a) `check_valid_vote_request`

   assert(msg.term ≥ current_term)
   assert(voted_for == NULL **or** voted_for == msg.candidate_pid)
   assert(msg.candidate log is up-to-date)

   (b) `post_valid_vote_request`

   current_term = max(current_term, msg.candidate_term)
   voted_for = msg.candidate_pid
   current_leader = msg.candidate_pid
   election_timeout = current_time + ELECTION_TIMEOUT

9. Receiving a VoteResponse from a peer (**handle_vote_response**)

   (a) `pre_handle_vote_response`

   assert(role == CANDIDATE)
   assert(msg.follower_pid **not in** votes_rcvd)

   (b) `post_handle_vote_response`

   votes_rcvd.add(msg)

10. Receiving a RequestAppend from a peer (`handle_append_entries_request`)

    (a) `handle_log_conflicts`

    if log[msg.prev_log_index].term ≠ msg.prev_log_term **then**
        var i = msg.prev_log_index
        while i < **length**(log)
            log.remove(i)
        return **true**

    if not handle_log_conflicts **then**
        log.append(msg.entries)
        commit_index = **min**(msg.leader_commit_index, **length**(log))

11. Receive ResponseAppend from a peer (`handle_append_entries_response`)

    (a) `pre_handle_append_entries_response`

    assert(role == LEADER)
    assert(msg.follower_term ≤ current_term)

    (b) `post_handle_append_entries_response`

    if msg.success == FALSE **then**
        next_index[msg.follower_pid] = **max**(next_index[msg.follower_pid] - 1, 0)
    else
        next_index[msg.follower_pid] = msg.match_index + 1
    match_index[msg.follower_pid] = msg.match_index

7 Rust Implementation

The initial goal of the implementation was to reuse as much code as possible from the `Single--Decree Paxos` implementation, but this proved difficult to accomplish since `Single-Decree Paxos` has no notion of log nor log-replication.

For the **Rust** implementation, the following implementation choices were made:

- Each peer is simulated by a single thread - therefore it processes messages received in a sequential order (does not process messages concurrently)
- The message exchanges are made usign the std::sync::mpsc module
- Includes an adjustable probability for a message to be "lost" - for the tests below, that probability was set to 10%
- To better test concurrent scenarios, a random delay of 1 to 100 milliseconds is included before sending any message
- Includes the optimization to send multiple entries in a single `RequestAppend` message
- Includes a 5% chance for a leader to fail for 10 seconds before sending an heartbeat

The Rust implementation uses the same state variable names as referred in **.2** and the function names for each step are identical to the ones presented in **.6**.

8 Tests and Results

For the testing scenario there is a client sending a new `RequestOperation` message every second, up to a total of 100 requests.

Up to 75 peers, the test results are as expected. The peers can handle leader failures and log conflicts, and the result of the protocol is a replicated log where all peer's apply the same operations in the same order. Note that if a leader fails while "holding" uncommitted entries, it is possible for those entries to be lost.

For a larger number of peers, the values for the election and heartbeat timeouts need to be adjusted to account for the fact that sending messages to all peers and waiting for their responses takes significantly longer, and if the timeouts aren't adjusted the protocol will end up in an infinite cycle of timeouts.

It was hard to get any measure of performance with the current implementation and testing environment for the following reasons:

1. Since the implementation uses the optimization that allows leader's to send multiple entries at the same time, the client needs to time his requests, otherwise the leader will send all entries at once and we don't obtain a realistic trace of how the protocol performs
2. Sending requests every X seconds is good to see how the protocol handles leader failures, but it makes it hard to obtain measurements, such as the average time elapsed to commit an entry

Despite that, the average amount of time taken to commit an entry, from the moment it was added to the log, while varying the number of peers involved in the protocol is tabled below:

	10 peers	25 peers	50 peers	75 peers
Elapsed Time	115ms	212ms	316ms	473ms

Note that these averages consider that the leader was receiving the entries exactly every 1 second, and only consider the entries received and committed by the same leader, not accounting for leader changes which take a significant amount of time. Therefore, these are not a good indicator of the implementation's performance.

9 Sources

In Search of an Understandable Consensus Algorithm (Extended Version), Diego Ongaro & John Ousterhout, https://raft.github.io/raft.pdf

Formal TLA+ specification for the Raft consensus algorithm, Diego Ongaro, https://github.com/ongardie/raft.tla