Raft Algorithm : Beyond the Lectures

CS 542 Course Project
Dhanesh V. - 210101117, **Ketan S.** - 210101118, **Shivam A.** - 210101119

---

1 Introduction and Motivation

We encountered the Raft algorithm before its coverage in our lectures and were captivated by its simplicity, inspiring us to implement it. While the lecture provided a foundational understanding of the Raft algorithm, it did not delve into certain aspects, such as membership changes and specific optimizations. As a result, we chose to pursue this as a course project to gain a deeper comprehension of Raft by implementing it comprehensively from scratch, adhering to industrial standards.

This report focuses on the modifications and analyses we conducted and the practical implementation details, omitting the standard Raft algorithm concepts already discussed in class. The following sections will detail our enhancements and implementation specifics.

2 Implementation Details

We have implemented raft protocol with leader election, log replication, multi-client handling, and add and remove servers in `golang` language using other libraries as explained below.

Every important step in Raft is logged in a file as shown in Figure 1 for better debugging and explanation of the algorithm. The entire code is made public in github repository, which

```
2024/11/01 22:25:28.618126 Node ID: 1, State: Follower | Server is started
2024/11/01 22:25:29.195582 Node ID: 1, State: Follower | election timeout
2024/11/01 22:25:29.291276 Node ID: 1, State: PreCandidate | transitioned to pre-candidate state with currentTerm: 0
2024/11/01 22:25:29.291276 Node ID: 1, State: PreCandidate | sent RequestVote RPC to Node 3 with Term: 1, LastLogIndex: -1, LastLogterm: 0
2024/11/01 22:25:29.291276 Node ID: 1, State: PreCandidate | sent RequestVote RPC to Node 2 with Term: 1, LastLogIndex: -1, LastLogterm: 0
```

Figure 1: Sample Log Output for Node State Transitions

can be found at https://github.com/botketan/Raft-Implementation. We used github as version control system as it enables collaboration among a team seamlessly.

2.1 Why golang

- `golang` can support **light weight** threads called `goroutines`, which make managing multiple tasks (like Raft's leader election, log replication, and state synchronization) straightforward and efficient without heavy threading.

- We need good support for locking mechanisms to make the different steps in raft algorithm thread-safe. Unlike languages like C++, where locking requires a lot of boilerplate code, in `golang`, it is just a single line to lock and unlock.
- `golang` integrates well with RPC systems like `gRPC` which we used for doing RPC calls.
- In industry also, `golang` is widely used to code distributed systems, hence to gain some expertise regarding the same, we chose this language.

2.2 MongoDB and gRPC

- Raft algorithm relies on fault-tolerant disk storage to store `votedFor`, `currentTerm`, `nodeID`, `log` etc which will allow it to wake up from crash. Since `golang` has good compatibility with NoSQL database `MongoDB`, we used it to store the variables every time it changed.
- The core part of Raft is Remote Procedural Calls (RPC); given the popularity of `gRPC` by Google for RPCs, we used it for the five RPCs - `AppendEntries`, `SubmitOperation`, `RequestVote`, `AddServer`, `RemoveServer`.
- `gRPC` handles the internal RPC stub while allowing us to focus on just the logic of the RPC calls by specifying the RPC definitions as **Protocol buffers**.

2.3 Key Value Store

- Raft can be used for any replicated state machines; we made an interface called `FSM` in which all that has to be done is to write the logic for applying the operation to the State machine.
- We have used a basic HashMap as Key Value store taking care of requests out of order.
- Since HashMap KV Store is volatile, `lastApplied` need not be stored in `MongoDB` as there won't be any problem reapplying the entries from log.
- However, if the KV store is non-volatile like `Sqlite` Database, we need to store the `lastApplied` variable also in `MongoDB` as they have to be applied only once.

2.4 Concurrency mechanisms

- We have used `mutex` lock in `golang` for concurrency among `AppendEntries`, `RequestVote` and other operations to avoid racing condition.
- `waitgroup` is used as a barrier to run the long running loops in Raft algorithm like `heartbeat` loop, `electionTimeout` loop, `commit` and `append` loops which `Wait` on the locks that are `broadcast` to wake them up at right moments.

3 Probabilistic Analysis of Leader Election

This section analyses the expected time for the election to timeout if no split vote occurs. Since we are choosing random election timeout values for every raft node, the probability of occurrence of a split vote is very low, which is taken as 0 for the case of analysis.

Assume that the leader crashes **after** sending heartbeats to all the nodes. Let there be r servers available after a leader failure, with their timeouts $t_1, t_2, \cdots, t_r$ chosen independently from $U[a, b]$. A candidate that times out first will start the election procedure.

- Let $M = \min(t_1, t_2, \cdots, t_r)$ be an RV that denotes the minimum timeout among remaining servers, such that $\mathbb{E}[M]$ gives the expected time for an election timeout to occur after a leader crashes.
- The $\mathbf{CDF}(M) = \mathbb{P}(M \leq x) = 1 - \prod_{i=1}^{r} \mathbb{P}(t_i > x) = 1 - \left(\frac{b-x}{b-a}\right)^r$.
- Thus, $\mathbf{PDF}(M):\ f_M(x) = \frac{d}{dx}\mathbb{P}(M \leq x) = \frac{r(b-x)^{r-1}}{(b-a)^r}$.
- Hence, $\mathbb{E}(M) = \int_a^b x\, f_M(x)\, dx = \frac{ar+b}{1+r}$. Thus, the election timeout is expected to occur $\frac{ar+b}{1+r}$ seconds after a leader crashes.
- Therefore, the time to election completion is given by

  $\mathbb{C}_t =$ Base election timeout $+ M +$ Time to send and receive Request Vote RPC $\pm$ error

  So $\mathbb{E}[\mathbb{C}_t] = a + \mathbb{E}[M] + 2 \times \mathbb{E}[N] \pm \epsilon$. Here

3.1 $\mathbb{E}[\mathbb{C}_t]$ for our implementation

We generated election timeouts randomly from a uniform distribution $U[300\,\text{ms}, 600\,\text{ms}]$, while the one-way network delay is approximated to be 0.7 ms. By applying the formula for $\mathbb{E}[\mathbb{C}_t]$ and assuming we have four available servers, we obtain the expected value as

$$\mathbb{E}[\mathbb{C}_t] = 676.4\,\text{ms} \pm \epsilon.$$

Therefore, in the case of no split votes, we anticipate that the cluster will be unavailable for approximately 676.4 ms, which is relatively low.

4 Improvements

This section elucidates the minor improvements that we had carried out in Raft algorithm to optimise and improve normal operations and to increase the availability of the cluster.

4.1 Pre-Vote Phase

The Raft consensus algorithm is widely utilized to manage distributed consensus in replicated state machines. However, high network instability or node isolation can lead to excessive term increases and leadership transitions, thereby impacting the stability of the Raft cluster. To address these issues, some implementations incorporate a *pre-vote phase*, which serves to minimize unnecessary term increments and avoid redundant elections, ultimately enhancing cluster stability.

4.1.1 Mechanism of the Pre-Vote Phase

In the standard Raft protocol, when a follower fails to receive heartbeats from the leader within a specified timeout period, it increments its term and initiates a new election. This straightforward approach can, however, result in term "bouncing" or a split-brain scenario, especially in situations where followers are temporarily isolated or experiencing delays. The pre-vote phase is designed to mitigate these effects by allowing nodes to verify their chances of election success before incrementing their term and formally starting an election.

4.1.2 Pre-Vote Request

When a node suspects that the current leader is down and considers initiating an election, it first enters the pre-vote phase. In this phase, the node does not immediately increment its term. Instead, it broadcasts *pre-vote requests* to other nodes in the cluster, effectively inquiring whether they would support an election if one were to be initiated. This is accomplished by sending out requests asking, "If I were to start an election, would you consider voting for me?"

4.1.3 Decision Process Based on Responses

Upon receiving the pre-vote requests, the peers respond based on their current term and state. The initiating node then evaluates these responses:

- If the node receives a majority of positive responses, indicating that the majority would consider it for leadership, it then increments its term and formally initiates an election.
- If it fails to secure a majority, the node remains in the follower role without starting an election. This approach avoids unnecessary term increments and potential disruption in the leadership.

4.1.4 Advantages of the Pre-Vote Phase

The inclusion of a pre-vote phase in Raft offers several advantages:

- **Reduction in Term Increments:** By avoiding premature term increments, the pre-vote phase reduces the likelihood of unnecessary leadership transitions, thereby improving cluster stability.
- **Increased Cluster Stability:** Particularly in clusters with high latency or transient network partitions, the pre-vote phase helps prevent unnecessary elections and mitigates split-brain scenarios.
- **Protection Against Partitioned Servers:** The pre-vote phase prevents a partitioned server from disrupting the cluster when it rejoins. The server cannot increment its term during its partition, as it cannot receive majority permission. When it rejoins, it still cannot increment its term, as the other nodes will have received heartbeats from the leader. Upon receiving a heartbeat from the leader, the partitioned server automatically returns to the follower state in the current term, thus avoiding unnecessary elections.

4.1.5 Addressing liveness

Consider an example where we have five servers. Servers 1, 2, and 3 are connected, server 4 is connected only to server 2, and server 5 has failed. Initially, server 4 was elected as the leader before it became partitioned from most of the cluster, connecting it only to server 2. Since server 4 is now isolated from the majority of servers, it is unable to make progress, as it

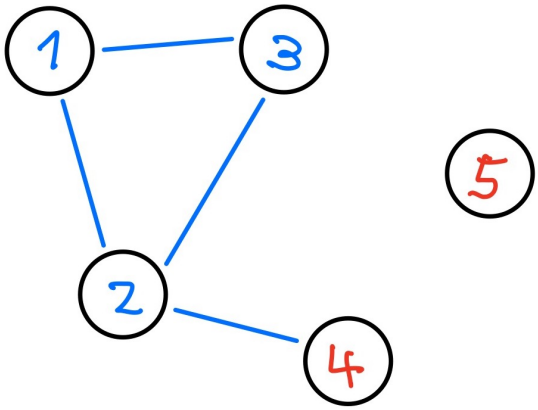


Figure 2: Example

cannot reach a quorum. Meanwhile, servers 1 and 3 will not receive heartbeats from the leader, causing them to timeout and initiate the pre-vote phase. However, neither server 1 nor server 3 will complete their pre-vote phase because server 2, which still receives regular *AppendEntries* messages from the leader (server 4), will not cast a pre-vote for either of them. Consequently, the system remains in a stalemate: a new leader cannot be elected, and the current leader (server 4) cannot make progress.

To address this, a leader should be stepped down when it doesnt receive **AppendEntries** response from the majority of the cluster. In that case, the leader, Server 4, here will step down and the rest of the servers will proceed with the election.

4.2 Handling Client Operations

In a Raft-based distributed system, client interactions with the servers are designed to ensure operations are processed only by the current leader. This behavior is essential, as Raft requires a single, consistent leader to handle client commands. In this section, we discuss client interactions using the `SubmitOperationRequest` and `SubmitOperationResponse` protocol buffer contracts, along with the mechanism by which the Raft server informs clients of the leader's identity if the server receiving the request is not the leader.

4.2.1 Request and Response Structure

The `SubmitOperationRequest` message represents the client's request to submit an operation to the Raft cluster:

```
message SubmitOperationRequest {
    string client_id = 1;
    int64 seq_no     = 2;
    bytes operation  = 3;
}
```

- **client_id**: A unique identifier for the client submitting the request. This helps the Raft cluster track client-specific operations and ensures idempotency, allowing the system to identify repeated requests and avoid duplication.

- **seq_no**: A sequence number used to order requests from a client. By including a sequence number, the Raft cluster can enforce the correct ordering of operations for each client, ensuring consistency. We have stored this in `MongoDB` as when a client restarts again, the sequence numbers should be unique.
- **operation**: The actual operation to be performed, represented as a byte array to support various types of data payloads.

The server's response to this request is encapsulated in the `SubmitOperationResponse` message:

```
message SubmitOperationResponse {
    bool success   = 1;
    string message = 2;
}
```

- **success**: A boolean indicating whether the operation was successfully submitted to the cluster leader for processing.
- **message**: A string message providing additional information about the request outcome, such as error details or leader redirection information if applicable.

4.2.2 Leader Redirection Mechanism

In Raft, only the leader can accept and commit client operations. When a client sends a `SubmitOperationRequest` to a Raft server, the server first checks whether it is the current leader:

- **If the server is the leader**: It processes the client request directly by appending the operation to its local log and initiating the replication to followers.
- **If the server is not the leader**: It rejects the client's request and responds with the identity of the current leader.

This leader redirection mechanism is crucial for maintaining Raft's consensus rules, as it prevents non-leader nodes from making changes to the replicated state.

4.3 Cluster Membership Change

The basic raft algorithm lacked the capability of dynamic changes to the raft clusters. We introduced the feature of modifying cluster memberships by adding and removing servers one at a time, ensuring stability and simplicity. Below is a step-by-step breakdown of these operations:

4.3.1 Adding a Server

1. **Update Cluster Configuration**:
   The leader initiates the addition of a new server by updating the cluster configuration to include the new server in the consensus group. This change is recorded as a configuration entry in the log, informing other servers that the new server is now officially part of the cluster.

2. **Propagate the Configuration Change**:
   The leader appends the new configuration entry to its log and replicates this entry across the cluster, ensuring that all servers are aware of the updated configuration and acknowledge the new server's role in consensus decisions.

3. **Quorum Maintenance**:
   The new server is now counted as part of the quorum needed for consensus. However, the cluster maintains the ability to reach consensus with existing servers if the new server is temporarily unreachable, preventing any immediate impact on availability.

4. **Begin Normal Operations**:
   The new server is fully integrated and participates equally in log replication, voting, and other Raft operations. The new configuration becomes stable once all servers have accepted it.

4.3.2 Removing a Server

1. **Update Cluster Configuration**:
   To remove a server, the leader modifies the cluster configuration to exclude the server. This change is recorded as a configuration entry in the log, notifying the cluster of the server's exclusion.

2. **Propagate the Configuration Change**:
   The leader replicates the updated configuration entry to all remaining servers. Once this configuration is acknowledged, the removed server is no longer part of consensus decisions, and the cluster size is adjusted accordingly.

3. **Quorum Check**:
   Before removing the server, the leader ensures that the remaining servers form a majority capable of reaching consensus independently. This guarantees that the removal does not disrupt the cluster's ability to process requests or maintain log consistency.

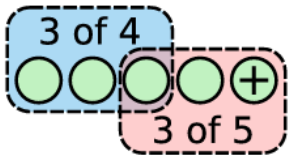
4. **Final Removal**:
   After replication, the removed server is no longer part of the quorum for future decisions. Raft continues normal operations with the new configuration, excluding the removed server.

4.3.3 Key Points for Safety and Availability

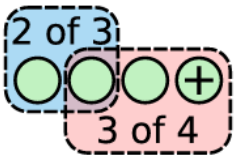
- **Sequential Changes**:
  By restricting changes to one server at a time, Raft simplifies state management, ensuring configuration changes are easy to track and verify. This avoids complex situations where multiple changes could lead to a loss of quorum or consistency issues.

- **Replication of Configuration Changes**:
  Both additions and removals of servers are handled as log entries, making it straightforward for Raft to manage and propagate configuration updates. This ensures consistent application of configuration changes across the cluster.

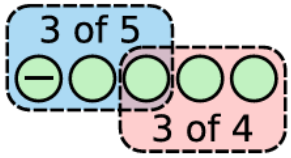
- **Majority Requirement**:
  Raft enforces that a majority quorum remains after any removal, preserving fault tolerance and allowing the system to handle network partitions or temporary server failures gracefully (See Fig. 3).



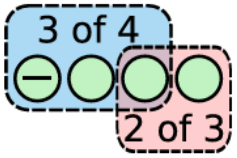
(a) Adding one server to a 4-server cluster.



(b) Adding one server to a 3-server cluster.



(c) Removing one server from a 5-server cluster.



(d) Removing one server from a 4-server cluster.

Figure 3: The image demonstrates the addition and removal of a single server from clusters of even and odd sizes while preserving majority overlap between the old and new clusters. In each subfigure, the blue box represents the majority of the original cluster, and the red box represents the majority of the new cluster after the change. When a server is added or removed, the majority overlap is maintained: the old and new clusters share enough servers to ensure that the new cluster's majority includes at least part of the original majority. This overlap is crucial for maintaining consistency and stability during membership changes in clustered systems.

4.3.4 Impact of Allowing Membership Changes in Raft

Initially, the Raft algorithm did not support dynamic membership changes, meaning that the cluster configuration was fixed, and any modification required a full restart or manual reconfiguration. By incorporating a mechanism to allow the addition and removal of servers during normal operation, we significantly enhance Raft's flexibility and adaptability. The key impacts of allowing membership changes are as follows:

- **Improved Scalability**: Allowing servers to be added to the cluster dynamically enables the system to scale in response to increased workload or demand. Additional servers can be incorporated to handle higher requests without interrupting ongoing operations, improving the system's responsiveness and scalability.
- **Enhanced Fault Tolerance and Reliability**: The ability to remove and replace failing or unreliable servers ensures the cluster can maintain a majority quorum even when individual nodes encounter issues. This dynamic reconfiguration allows Raft to maintain high availability and robust fault tolerance without requiring a full shutdown or reconfiguration.
- **Reduced Maintenance Overhead**: Previously, modifying the cluster required manual intervention and often a full restart, which increased operational complexity and potential downtime. With membership changes allowed, administrators can adjust the cluster configuration with minimal disruption, reducing the overhead of system maintenance.

4.4 Append Entries Optimization

In the Raft consensus algorithm, log replication typically involves the leader server identifying the most recent matching entry in the follower's log and sending any missing entries sequentially. When an inconsistency is found between the leader and follower logs, Raft reduces the

follower's next index step-by-step, attempting to find a match. This traditional approach requires exchanging lots of messages between the leader and follower in case of many missing or mismatching entries.

In our optimized approach, rather than decrementing the follower's next index by one with each inconsistency, the follower server includes the term of the conflicting entry and the first index it stores for that term and communicates this index to the leader. This allows the leader to bypass all the conflicting entries in that term, effectively requiring one Append Entries RPC for each term with conflicting entries. This method reduces the number of back-and-forth messages required to achieve log consistency, streamlining the append entries process and improving overall system efficiency.

This optimization minimizes the latency associated with log replication and ensures faster convergence in cases of network delay or server restarts, thereby improving the robustness of the Raft algorithm in dynamic environments.

5 Conclusion

Thus this project allowed us to gain deeper insight into raft which made us realise the complexities that arises when implementing an algorithm. It allowed us to learn and experiment various technologies like `gRPC`, `golang` etc. while serving as a good exercise for writing thread safe concurrent programmes. Thus by taking raft algorithm as an extension of lecture, we were able to gain more insights into the same.