Raft Consensus

Deepak HR
Computer Science and Engineering
IIT Dharwad
Dharwad, India
170010026@iitdh.ac.in

Rohan Shrothrium
Computer Science and Engineering
IIT Dharwad
Dharwad, India
170020031@iitdh.ac.in

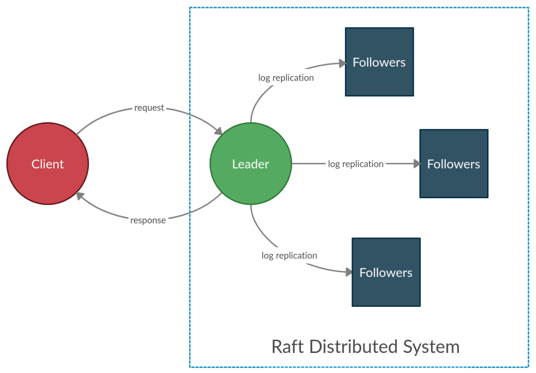


Fig. 1. Raft Architecture

***Abstract*—Raft is a consensus algorithm that offers a generic way to distribute a state machine across a cluster of computing systems, ensuring that each node in the cluster agrees upon the same series of state transitions. It offers many important features like Reliability, Replication, Redundancy, And Fault-Tolerance.**

I. Introduction

This report describes our work the Distributed Systems project during the Fall 2020 semester. We summarize the main sections of the consensus algorithm as explained in the Raft consensus paper [1]. We also discuss our implementation and tests. The sections can be summarized as follows.

A. Basics

Jargon related to the algorithm is discussed to paint a concise picture of different components of the system.

B. Leader Election

A new leader must be chosen at the start of the session or when an existing leader fails. II-B will help you understand how the leader election [2] happens in Raft Consensus.

C. Log Replication

The leader must accept log entries from clients and replicate them across the cluster so that all the nodes in the system have the same log entry and section II-C will help you understand how the log replication [3] happens in Raft Consensus.

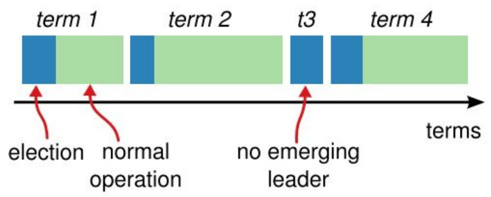


Fig. 2. Terms in Raft

D. Safety

The key safety property for Raft is the State Machine Safety Property, that is, if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index. Section II-D describes how Raft ensures this property; the solution involves an additional restriction on the election mechanism described in II-B.

E. Implementation and Testing

Our raft implementation using GoRPC and other test details are discussed in III.

II. Consensus Algorithm

A. Basics

A Raft cluster is an asymmetric system that consists of several servers. Each server at a particular time maybe one of three types as follows

1) Leader: Usually there is one leader and the rest of the servers are followers. Leaders respond to all client requests - if a client contacts a follower, the request is forwarded to the leader.

2) Follower: Followers are passive; Their only role is to respond to requests that come from leaders or candidates.

3) Candidate: Candidates are used to elect new leaders during leader election.

Time is divided into terms as shown in Fig. 2. The start of a term is marked by the leader election. A server winning the election would be the leader for the rest of the term. It is ensured that there is at-most one leader at any point in time. Terms act as a logical clock that helps servers to detect stale leader and stale requests.

Communication is done using remote procedural calls (RPCs). Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance.

B. Leader Election

Raft uses a concept called heartbeat mechanism to trigger leader election. The life-cycle of a node in the cluster is explained in Fig. 3. When a server starts, it starts as a follower. The leaders and followers communicate using RPCs and they use two RPC functions

- Request Vote()
- AppendEntries()

The leader sends a heartbeat tick to all the followers at least once in a given time interval irrespective of it getting a log entry from a client or not. This communication of the leader with the followers is done by the RPC call **AppendEntries()**. If a follower doesn't receive this heartbeat tick in the time interval, it starts the process of leader election. The steps are as follows:

- Increases it's term number.
- Votes for itself and issues request vote RPC in parallel to all servers in the cluster.

Once this is done, there are three possible outcomes: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.
A candidate becomes a leader if it receives majority votes in the cluster. Each server in the cluster can vote at most one candidate. Once a candidate wins the election, it becomes the leader and sends heartbeat to all the remaining servers.
While waiting for votes if a candidate receives a **AppendEntries()** RPC call from another server, it checks the term number of this server. If the term number is at least as big as the candidates term number, the candidate accepts this server as the leader and then becomes a follower and waits for the heartbeat. If the term number is lower than the candidate's term number, the candidate rejects the RPC and continues to be a candidate.
If two or more servers make a **RequestVote()** RPC call at the same time, there can be chance that no one receives majority vote and hence no leader is elected. Once this happens, each candidate has to wait for a certain period of time which is chosen randomly, typically between 150-300 ms, and then make a **RequestVote()** RPC call again. This waiting time is chosen randomly so that there is a lower probability of the vote split to happen again, If all the candidates start a vote again at the same time, it will end up in an infinite loop and no one will be elected as a leader.

C. Log Replication

The job of the leader is to servicing the client. Each client request contains a command to be executed by the replicated state machines. Once this request is received by the master, it sends a **AppendEnteries()** call to all the followers. Once all the followers append the entry, the leader applies the entry, runs the command and returns the request to the client. If any server fails, acknowledgement packer is dropped, or **AppendEnteries()** request is not received by any of the followers, it retries indefinitely until the followers store the entry.

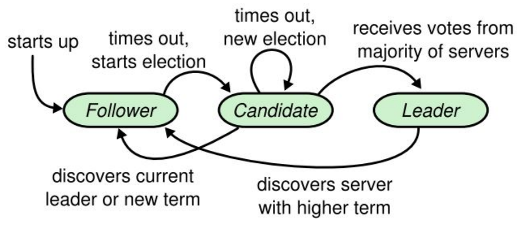

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Fig. 3. Leader election

Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created and a command for the state machine. An entry is considered committed if it is safe for that entry to be applied to state machines. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines.
A log entry is considered committed once the leader that created the entry has replicated it on a majority of the servers. It also commits all the previous entries. Raft maintains the following properties:

- If two entries in different logs have the same index and term, they have same command.
- If two entries in different logs have the same then the logs are identical in all preceding entries.

Until a leader crashes the followers' logs are consistent with the leader's logs, however, when the leader crashes some inconsistencies are bound to occur. These inconsistencies are handled by the master in the following way:

- For each follower it finds the latest log entry up till which both the logs are consistent.
- It deletes all the entries in the follower log after that point and that of the leader is sent.
- The master maintains a variable **nextIndex** for each follower which is the index of the next log entry the master will send to it's follower.
- When a new leader comes to power, it initializes the **nextIndex** to the latest log entry in it's local logs. If there are any inconsistencies the leader decrements the value of **nextIndex** and tries again until there are no inconsistencies.

By following this the master successfully replicates it's logs to all the followers.

D. Safety

Consensus among all server requires additional measures. The features mentioned till now do not prevent a follower that was unavailable for a while and missed a few log commits from winning and election and jeopardizing command sequences across different servers. We need to ensure that a leader satisfy the Leader Completeness Property from Fig. 4.

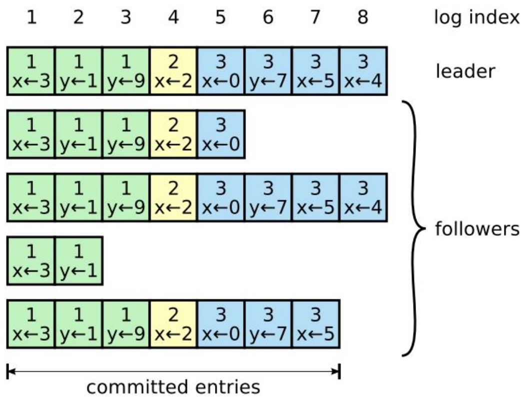


Fig. 4. Log replication

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a 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 the same index.

Fig. 5. Guarantees offered by raft

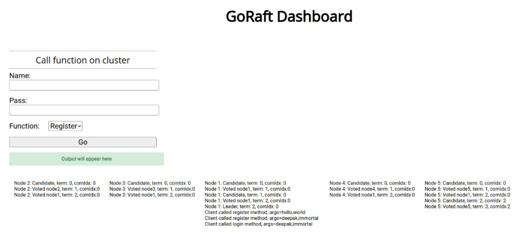
We discuss subtle points for committing logs here to ensure the same.

1) Election Restriction: It is important to ensure that the leader eventually stores all of the committed log entries. Raft guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. Elections are *restricted* to only those machines that have all the committed entries. This is implemented using the condition that if a follower receives a **RequestVote()** RPC and it sees that its logs are more up-to-date than the logs of the requesting candidate, it denies its vote. This is achieved using index and the term number.

2) Committing entries from previous terms:

- Only log entries from the leader's current term are committed by counting replicas.
- Once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property.

3) Follower and candidate crashes: If a follower or a candidate crashes, the raft retries the RPCs indefinitely. The crashed server will respond to these requests when it restarts. Raft RPCs are idempotent - if a follower receives **AppendEntries()** request containing logs that are already present in its log, it ignores those entries in the request.



```
Node 2: Candidate, term: 0, comIdx: 0
Node 2: Voted node2, term: 1, comIdx:0
Node 2: Voted node1, term: 2, comIdx:0

Node 3: Candidate, term: 0, comIdx: 0
Node 3: Voted node3, term: 1, comIdx:0
Node 3: Voted node1, term: 2, comIdx:0

Node 1: Candidate, term: 0, comIdx: 0
Node 1: Voted node1, term: 1, comIdx:0
Node 1: Candidate, term: 1, comIdx: 0
Node 1: Voted node1, term: 2, comIdx:0
Node 1: Leader, term: 2, comIdx: 0
Client called register method, args=hello,world
Client called register method, args=deepak,immortal
Client called login method, args=deepak,immortal

Node 4: Candidate, term: 0, comIdx: 0
Node 4: Voted node4, term: 1, comIdx:0
Node 4: Voted node1, term: 2, comIdx:0

Node 5: Candidate, term: 0, comIdx: 0
Node 5: Voted node5, term: 1, comIdx:0
Node 5: Voted node1, term: 2, comIdx:0
Node 5: Candidate, term: 2, comIdx: 2
Node 5: Voted node5, term: 3, comIdx:2
```

Fig. 6. GoRaft Dashboard for our Raft system implementation

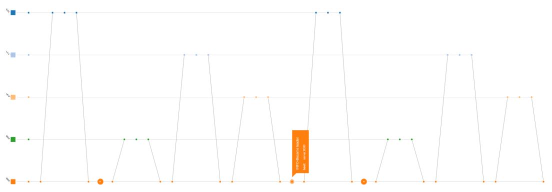


Fig. 7. Shiviz plot of a leader election in our system. The orange node is able to request votes from all other servers and gathers 4 votes. It escalates to a leader and sends its heartbeat to everyone else.

4) Timing and availability: Raft must guarantee safety independent of response times. If message exchanges take longer than the typical time between server crashes, candidates will not stay up long enough to win an election; without a steady leader, Raft cannot make progress. The following is a requirement for steady maintenance of a leader.

$$broadcastTime << electionTimeout << MTBF$$

BroadcastTime is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses; electionTimeout is the election timeout described in II-B; and MTBF is the average time between failures for a single server.

III. Implementation and Testing

We have implemented Raft as a part of a replicated state machine. For demonstration purposes, we use this system as a user database with register and login functions. We initialize a five node raft cluster hosted locally on different ports. A client has also been implemented to send **Register** or **Login** RPC requests to the cluster leader. The network topography is as shown in Fig. 1. The code is available on https://github.com/deepakhr1999/go_raft. Steps to install and run the code is discussed in the readme file. Implementation and testing is discussed feature by feature.

A. Dashboard

A dashboard (see Fig. 6) is also built to run in the browser. It enables a user to behave as a client and make **Register** or **Login** RPC requests to the cluster. The dashboard also streams the terminal output of all the nodes so we can understand the internal working of each node.

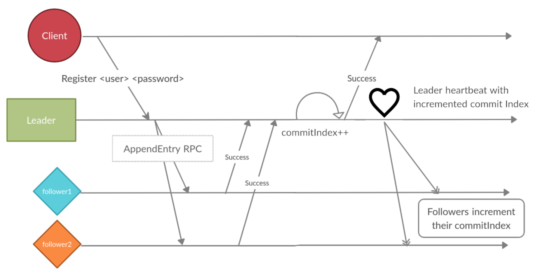

Client
Register <user> <password>
Success
Leader heartbeat with incremented commit Index
Leader
commitIndex++
AppendEntry RPC
Success
Success
follower1
Followers increment their commitIndex
follower2

Fig. 8. Timeline of RPCs made when a client registers a user at the raft leader node

B. Leader Election

Shiviz [4] logs of our cluster's leader election timeline has been visualized in Fig. 7. Since no leader heartbeat is detected, the first node requests and gathers vote from the followers and becomes the leader. It asserts its leadership by broadcasting its heartbeats across the cluster. We set the election timeout to be a random duration between 0-6 seconds in contrast to 100-150ms in the original paper. This is done so that we can see how the nodes escalate to the candidate state, gather votes and one of them becomes a leader in the dashboard.

Testing Leader Election:

- **Initial election:** We start the cluster and check that exactly one leader is elected.
- **Election after leader failure:** We kill the leader node and check that the remaining nodes elect a leader.

C. Persistence

This replicated state machine must be able to store its state on the disk. When the whole system is brought down and restarted, all the nodes must be able to resume normal operation. This is implemented by storing the commit index, latest vote and the logs array as a JSON file for each node. The node looks for and initializes its state from this file on restart.

Testing Persistence:

- **Check JSON file:** Start the cluster and register a few users. We verify that they have been added to all the nodes' JSON files.
- **Restart check:** We shutdown the whole system and then restart it. We verify that all users registered before the shutdown are authenticated.

D. Log Replication

When a user registers herself on the system, an entry containing the name and the password is appended to the leader log. The leader commits this entry when it receives success signal from majority of the followers. During the subsequent leader heartbeats, the incremented commit index is shared with the followers. This process timeline is shown in Fig.8. When the same user sends a **Login** request, the credentials are searched through the leader log.

Testing Log Replication:

- **Check JSON file:** Start the cluster and register a few users. Verify they have been added to the node's JSON file.
- **Safety check:** We kill the leader node and check that all registered users are authenticated by the system with the new leader.

IV. Conclusion

The Raft algorithm is designed to be understandable while achieving the all primary goals - correctness and efficiency. We can see that lack of understanding can hinder a developer from bringing the features in the published paper to practice. This project has been a journey of understanding a very popular consensus algorithm widely used in distributed systems.
Implementing the raft protocol from scratch gave us an insight into developing distributed systems. All the programs we had written prior to this project were synchronous single system programs. In practice, any worthwhile large system is meant to be scalable and inevitably distributed. We were pleased to behold the role and beauty of asynchronous execution in distributed systems. We look forward to perform all the tests mentioned above, during the project demonstration and replicate the joy of completing a distributed systems project.

V. Acknowledgements

- We would like to thank Prof. Kedar Khandeparkar for giving us this opportunity to build a worthwhile system from the ideas we learnt in class.
- We have directly taken the figures from the original Raft paper [1].
- Figs 1 and 8 have been drawn for illustration on https://creately.com/

References

[1] In Search of an Understandable Consensus Algorithm
[2] https://en.wikipedia.org/wiki/Leader_election
[3] https://en.wikipedia.org/wiki/Replication_(computing)
[4] Shiviz: A visualization engine that generates interactive communication graphs from distributed system execution logs.