Raft

Johannes Jørgensen (jgjo),
Kevin Skovgaard Gravesen (kegr),
Joakim Andreasen (joaan)

December 1, 2024

Introduction

The following report describes the Implementation of the Raft consensus algorithm in Go made by Github User @jmsadair.
@jmsadair's implementation of the Raft consensus algorithm implements both leader elections with votes and node states, log replication via. gRPC and snapshots of the state if the logs.

This report will go through the @jmsadair's implementation and discuss the diffrent parts of the implementation, with focus on the quality of the implementation and Golang features.

Source: Github Repository - Raft @jmsadair.

Go features

The raft implementation is written using go routines and some channels. Some examples of the go routines being used could be the appending and commiting of information to the log, election handling and heartbeats. The reason for the use of channels is to move the decisions and actions take by the goroutines to the main routine. By combining these go features as much of the Raft implementation is multithreaded as possible.

Go's defer keyword is also used but, besides the routine and channel usage, the rest of the go features is built in go quality of life features. Which is Go's spin on features from other languages, such as slices.

Node communication

@jmsadair uses gRPC as the method of communication between the nodes in the cluster. The gPRCs include the the ability of a node to 1. request missing log enteries from the leader 2. Request votes from other nodes in the cluster in a leader election.

The transport.go file handles the communication used by a node in the cluster, both to send and receive RPCs. The transporter converts the requests and responses to the right protobuf instances and sends them to the right node in the cluster - This happens both for sendVote and log entries.

Implementation quality

The quality of the program is high, primarily due to the safety checks through the program. The implementation also uses tests to ensure the program works as intended.

Election safety

A constant running "electionTicker" go routine is being send to sleep with a random amount of time. This is to ensure that no two nodes are asking for an election at the same time. The "mutex" structure is also used for atomic locks which ensures that each node's internal decisions can be trusted. And that a node for example is not both accepting another node to be elected and voting on it self at the same time.

Leader Append-Only

Multiple checks are in place to ensure that only the leader could append to the log. That can, among other places, be seen in the ”sendAppendEntries” function that send new entries to the other nodes. This function have a check in it that prevents to node from sending anything if it is not the leader. With this even a wrongly use of the method won’t cause an issue in this case, even tho it should be rectified.

The majority requirement of adding to the log is also implemented, which Raft requires to ensure that the leader is not deciding on decisions without the other nodes following along.

Log Matching

Log Matching is the concept of maintaning consistency across the logs in the cluster of nodes. The Implementation that @jmsadair gurrentee log matching with log replication.

The ’AppendEntries()’ method handles log replication requests from the leader. It takes a request that contains the data of the log entry to replicate, the index and term of the current and previous entry. If everything goes as planed, the log entry is appended to the log of the node. If something goes wrong, like network partitions, out-of-date terms or invalid log indexes, the method will reject the request - or it will become a follower if the request has a more up to date term.

Leader completeness

To ensure a leader is either active or being actively selected, without ending in deadlock, the implementation uses a randomizer that delays the vote request for a random amount of time (default 300ms and 600ms). The randomizer is implemented to randomize who will send the request for votes first and therefore the chances of the same leader being chosen every program restart. The program also uses majority vote system, to mitigate the chance of a deadlock when there are two nodes getting votes. If there is two nodes that get equal amount of votes, the nodes resets the election and sends new requests for votes. Then due to the randomizer, the chances of a non-majority vote happens diminishes for every election reset that occurs.

State Machine Safety

Source