Report on the Hashicorp Raft Implementation

ghej, luvr, rono @ ITU, Group "G.L.R"

December 3, 2024

Contents

Links

- Hashicorp Raft - `https://github.com/hashicorp/raft`
- Example application - `https://github.com/Jille/raft-grpc-example`
- Example gRPC Transport implementation - `https://github.com/Jille/raft-grpc-transport/`

1 Go-specific features

1.1 Goroutines

In the chosen implementation, goroutines are employed in various ways. In `raft.go`, the `run()` method creates a thread to keep the program running. Then, in `api.go` the `NewRaft` method starts `run()` as a goroutine. It also starts separate goroutines for the FSM, and Snapshot service.

Goroutines are used extensively to run remote procedure calls. Once a listener is connected and is serving a registered gRPC service, incoming calls will create goroutines to run the procedures defined by the server interface. This interface defines a set of methods for responding to remote procedure calls.

1.2 Channels

Channels are fundamental to asynchronous routines Go. Fundamentally these are FIFO blocking queues allowing routines to send and receive data.

In the chosen implementation, channels are used in various locations to provide means of communication. In the `api.go` of the Raft library, Raft nodes utilize several channels, for a variety of purposes, including:

- **applyCh**, which sends logs to the main goroutine for commitment and updates to the FSM(finite state machine).
- **leaderCh**, which is used to communicate leadership changes between nodes.
- **verifyCh**, which verifies with the main goroutine that the node is still the leader.
- **rpcCh**, which allows a node to receive messages from other nodes in the cluster. The HashiCorp implementation has a Transport interface, which is then augmented by Jilles implementation using grpc(*See Communication methods*).

1.3 sync package

In the chosen implementation, various `sync` components are used to ensure the safety and liveness of the program. In addition to its basic components that we have previously used in our own work, the `RWMutex` is used extensively. This allows for the shared-read/exclusive-write interlocking also commonly seen on database systems, improving overall concurrency.

2 Communication methods

As seen in `transport.go` of the Raft library, the `Transport` interface defines the communications used to implement Raft. The RPC's for Raft include requests to append entries, calls for voting, timeout announcements. These have been included in the interface as `AppendEntries, RequestVote & TimeoutNow`.

Some additions and enhancements have later been added as separate interfaces, in order to not break existing use of the library:

1. `WithPreVote` solves a robustness issue that is well described here: `https://dev.to/tarantool/raft-notalmighty-how-to-make-it-more-robust-3a11`

2. `WithClose` can be added to allow nodes to shutdown cleanly.

Included with the example application by Jille was an example of how to satisfy the Transport interface with gRPC, including the added pre-vote feature. In addition to this, the example shows how one must separately implement some service using the log. The example application logs the three longest words that were sent to it. Any of the peers may receive words and can change the longest words retained, with the state being *eventually consistent* among the set of peers.

3 Properties guaranteed by the protocol implementation

The protocol as described in the `README` of the Raft library follows closely what we have previously seen at the lecture. Instead of logging a series of messages indefinitely however, the log allows the change of some shared states (three words) in such a way, that the system agrees on the order in which changes happened. Initially we imagined the log aspect of Raft as a message log, though this is also possible to implement.

To avoid issues with an unbounded log, where a new follower would have to parse a log of infinite size to restore to the current state, a system of snapshots is deployed. This is to minimize the work done to restore a follower to the current state. It is assumed that implementations using this library will have some upper limit on the logged information.

This implementation of raft is also able to handle servers joining and leaving, so long as a quorum is available to agree on the new peer set. This part of the implementation is robust to a failure of $\frac{N}{2} - 1$ servers, with N being the quorum size.