Raft Structure Advice

A Raft instance has to deal with the arrival of external events (Start() calls, AppendEntries and RequestVote RPCs, and RPC replies), and it has to execute periodic tasks (elections and heart-beats). There are many ways to structure your Raft code to manage these activities; this document outlines a few ideas.

Each Raft instance has a bunch of state (the log, the current index, &c) which must be updated in response to events arising in concurrent goroutines. The Go documentation points out that the goroutines can perform the updates directly using shared data structures and locks, or by passing messages on channels, perhaps to a central goroutine that manages state changes. Either can be made to work well, but we believe that for Raft it is most straightforward to use shared data and locks.

A Raft instance has two time-driven activities: the leader must send heart-beats, and others must start an election if too much time has passed since hearing from the leader. It's probably best to drive each of these activities with a dedicated long-running goroutine, rather than combining multiple activities into a single goroutine.

The management of the election timeout is a common source of headaches. Perhaps the simplest plan is to maintain a variable in the Raft struct containing the last time at which the peer heard from the leader, and to have the election timeout goroutine periodically check to see whether the time since then is greater than the timeout period. It's easiest to use time.Sleep() with a small constant argument to drive the periodic checks; time.Ticker and Time.Timer are difficult to use correctly.

You'll want to have a separate long-running goroutine that sends committed log entries in order on the applyCh. It must be separate, since sending on the applyCh can block; and it must be a single goroutine, since otherwise it may be hard to ensure that you send log entries in log order. The code that advances commitIndex will need to kick the apply goroutine; it's probably easiest to use a condition variable (Go's sync.Cond) for this.

Each RPC should probably be sent (and its reply processed) in its own goroutine, for two reasons: so that unreachable peers don't delay the collection of a majority of replies, and so that the heartbeat and election timers can continue to tick at all times. It's easiest to do the RPC reply processing in the same goroutine, rather than sending reply information over a channel.

Keep in mind that the network can delay RPCs and RPC replies, and when you send concurrent RPCs, the network can re-order requests and replies. Figure 2 is pretty good about pointing out places where RPC handlers have to be careful about this (e.g. an RPC handler should

handlers have to be careful about this (e.g. an RPC handler should ignore RPCs with old terms). Figure 2 is not always explicit about RPC reply processing. The leader has to be careful when processing replies; it must check that the term hasn't changed since sending the RPC, and must account for the possibility that replies from concurrent RPCs to the same follower have changed the leader's state (e.g. nextIndex).