## III Raft

Answer each of these questions with reference to the Raft protocol described in the paper's Figure 2 (without membership change, without log compaction, and without Section 8).

**6. [7 points]:** Suppose the leader waited for **all** servers (rather than just a majority) to have $matchIndex[i] >= N$ before setting commitIndex to $N$ (at the end of Rules for Servers). What specific valuable property of Raft would this change break?

**Answer:** Raft would no longer be available (would no longer make progress) if there were even a single crashed or unreachable server.

**7. [7 points]:** Suppose the leader immediately advanced commitIndex to its last log entry without waiting for responses to AppendEntries RPCs, and regardless of the values in matchIndex. Describe a specific situation in which this change would lead to incorrect execution of the service being replicated by Raft.

**Answer:** If less than half of the followers received and accepted an AppendEntries RPC, then the leader could commit an entry that could be lost during a subsequent leader change. This might cause the original leader to respond positively to the client even though the request's effects might disappear. In addition the leader would not be able to correctly convert to follower, since its application state (e.g. k/v table) would reflect a non-existent operation.

**8. [7 points]:** There are five Raft servers. S1 is the leader in term 17. S1 and S2 become isolated in a network partition. S3, S4, and S5 choose S3 as the leader for term 18. S3, S4, and S5 commit and execute commands in term 18. S1 crashes and restarts, and S1 and S2 run many elections, so that their currentTerms are 50. S3 crashes but does not re-start, and an instant later the network partition heals, so that S1, S2, S4, and S5 can communicate. Could S1 become the next leader? How could that happen, or what prevents it from happening? Your answer should refer to specific rules in the Raft paper, and explain how the rules apply in this specific scenario.

**Answer:** No. S1 must get a vote from one of S4 and S5 to get an election majority, and both S4 and S5 must be aware of operations committed in term 18, since they were both part of S3's bare majority. S1's log must end with an entry no later than term 17, since it would not have been able to win an election for any later term. By the leader election restriction rule, S4 and S5 won't vote for S1 since S4 and S5's logs end with entries that have term 18.

## IV Lab 2

Ben's Raft is failing the Lab 2 tests, which say that log entries are appearing on the apply channel out of order. Here is the part of his code that sends on the apply channel. Ben's implementation calls this function every time it updates the commit index. There is a single lock (Go mutex) protecting each Raft instance, and the calling code always holds the lock.

```
func (rf *Raft) applyToService() {
 for rf.commitIndex > rf.lastApplied {
   entry := rf.log[rf.lastApplied + 1]
   msg := ApplyMsg {
     Index: rf.lastApplied + 1,
     Command: entry.Command,
   }
   rf.lastApplied++
   go func() { rf.applyCh <- msg } ()
 }
}
```

**9. [7 points]:** Explain Ben's bug.

**Answer:** Ben is sending to rf.applyCh in a goroutine. If there are multiple such goroutines running (which can happen when multiple goroutines are spawned in that loop in a single call to applyToService), nothing prevents them from completing the sends out of the intended order (there's no explicit synchronization between the goroutines to ensure that happens).