6.824 2017 Lecture 5: Raft (1)

why are we reading this paper?
  distributed consensus is a hard problem that people have worked on for decades
  Lab 2 and 3 are based on Raft

this lecture
  today: Raft elections and log handling (Lab 2A, 2B)
  next: Raft persistence, client behavior, snapshots (Lab 2C, Lab 3)

overall topic: fault-tolerant services using replicated state machines (RSM)
  [clients, replica servers]
  example: configuration server, like MapReduce or GFS master
  example: key/value storage server, put()/get() (lab3)
  goal: same client-visible behavior as single non-replicated server
    but available despite some number of failed servers
  strategy:
    each replica server executes same commands in same order
    so they remain replicas (i.e., identical) as they execute
    so if one fails, others can continue
    i.e. on failure, client switches to another server
  both GFS and VMware FT have this flavor

a critical question: how to avoid split brain?
  suppose client can contact replica A, but not replica B
  can client proceed with just replica A?
  if B has really crashed, client *must* proceed without B,
    otherwise the service can't tolerate faults!
  if B is up but network prevents client from contacting it,
    maybe client should *not* proceed without it,
    since it might be alive and serving other clients -- risking split brain

example of why split brain cannot be allowed:
  fault-tolerant key/value database
  C1 and C2 are in different network partitions and talk to different servers
  C1: put("k1", "v1")
  C2: put("k1", "v2")
  C1: get("k1") -> ???
  correct answer is "v2", since that's what a non-replicated server would yield
  but if two servers are independently serving C1 and C2 due to partition,
    C1's get could yield "v1"

problem: computers cannot distinguish crashed machines vs a partitioned network
  both manifest as being unable to communicate with one or more machines

We want a state-machine replication scheme that meets three goals:
  1. remains available despite any one (fail-stop) failure
  2. handles partition w/o split brain
  3. if too many failures: waits for repair, then resumes

The big insight for coping w/ partition: majority vote
  2f+1 servers to tolerate f failures, e.g. 3 servers can tolerate 1 failure
  must get majority (f+1) of servers ti agree to make progress
    failure of f servers leaves a majority of f+1, which can proceed
  why does majority help avoid split brain?
    at most one partition can have a majority
  note: majority is out of all 2f+1 servers, not just out of live ones
  the really useful thing about majorities is that any two must intersect
    servers in the intersection will only vote one way or the other
    and the intersection can convey information about previous decisions

Two partition-tolerant replication schemes were invented around 1990,
  Paxos and View-Stamped Replication
  in the last 10 years this technology has seen a lot of real-world use
  the Raft paper is a good introduction to modern techniques

*** topic: Raft overview

state machine replication with Raft -- Lab 3 as example:
  [diagram: clients, 3 replicas, k/v layer, raft layer, logs]
  server's Raft layers elect a leader
  clients send RPCs to k/v layer in leader
    Put, Get, Append
  k/v layer forwards request to Raft layer, doesn't respond to client yet
  leader's Raft layer sends each client command to all replicas
    via AppendEntries RPCs

    each follower appends to its local log (but doesn't commit yet)
    and responds to the leader to acknowledge
  entry becomes "committed" at the leader if a majority put it in their logs
    guaranteed it won't be forgotten
    majority -> will be seen by the next leader's vote requests for sure
  servers apply operation to k/v state machine once leader says it's committed
    they find out about this via the next AppendEntries RPC (via commitIndex)
  leader responds to k/v layer after it has committed
    k/v layer applies Put to DB, or fetches Get result
  then leader replies to client w/ execution result

why the logs?
  the service keeps the state machine state, e.g. key/value DB
    why isn't that enough?
  it's important to number the commands
    to help replicas agree on a single execution order
    to help the leader ensure followers have identical logs
  replicas also use the log to store commands
    until the leader commits them
    so the leader can re-send if a follower misses some
    for persistence and replay after a reboot (next time)

there are two main parts to Raft's design:
  electing a new leader
  ensuring identical logs despite failures

*** topic: leader election (Lab 2A)

why a leader?
  ensures all replicas execute the same commands, in the same order

Raft numbers the sequence of leaders using "terms"
  new leader -> new term
  a term has at most one leader; might have no leader
  each election is also associated with one particular term
    and there can only be one successful election per term
  the term numbering helps servers follow latest leader, not superseded leader

when does Raft start a leader election?
  AppendEntries are implied heartbeats; plus leader sends them periodically
  if other server(s) don't hear from current leader for an "election timeout"
    they assume the leader is down and start an election
  [state transition diagram, Figure 4: follower, candidate, leader]
  followers increment local currentTerm, become candidates, start election
  note: this can lead to un-needed elections; that's slow but safe
  note: old leader may still be alive and think it is the leader

what happens when a server becomes candidate?
  three possibilities:
    1) gets majority, converts to leader
      locally observes and counts votes
      note: not resilient to byzantine faults!
    2) fails to get a majority, hears from another leader who did
      via incoming AppendEntries RPC
      defers to the new leader's authority, and becomes follower
    3) fails to get a majority, but doesn't hear from a new leader
      e.g., if in minority network partition
      times out and starts another election (remains candidate)
  note: in case (3), it's possible to keep incrementing the term
    but cannot add log entries, since in minority and not the leader
    once partition heals, an election ensues because of the higher term
    but: either logs in majority partition are longer (so high-term
      candidate gets rejected) or they are same length if nothing happened
      in the majority partition (so high-term candidate can win, but no damage)

how to ensure at most one leader in a term?
  (Figure 2 RequestVote RPC and Rules for Servers)
  leader must get "yes" votes from a majority of servers
  each server can cast only one vote per term
  at most one server can get majority of votes for a given term
    -> at most one leader even if network partition
    -> election can succeed even if some servers have failed

how does a new leader establish itself?
  winner gets yes votes from majority
  immediately sends AppendEntries RPC (heart-beats) to everybody
    the new leader's heart-beats suppress any new election

an election may not succeed for two reasons:
  * less than a majority of servers are reachable
  * simultaneous candidates split the vote, none gets majority

what happens if an election doesn't succeed?
  another timeout (no heartbeat), another election
  higher term takes precedence, candidates for older terms quit

how to set the election timeout?
  each server picks a random election timeout
    helps avoid split votes
  randomness breaks symmetry among the servers
    one will choose lowest random delay
    avoids everybody starting elections at the same time, voting for themselves
  hopefully enough time to elect before next timeout expires
  others will see new leader's AppendEntries heartbeats and
    not become candidates
  what value?
    at least a few heartbeat intervals (network can drop or delay a heartbeat)
    random part long enough to let one candidate succeed before next starts
    short enough to allow a few re-tries before tester gets upset
      tester requires election to complete in 5 seconds or less

*** topic: the Raft log (Lab 2B)

we talked about how the leader replicates log entries
  important distinction: replicated vs. committed entries
    committed entries are guaranteed to never go away
    replicated, but uncommitted entries may be overwritten!
  helps to think of an explicit "commit frontier" at each participant

will the servers' logs always be exact replicas of each other?
  no: some replicas may lag
  no: we'll see that they can temporarily have different entries
  the good news:
    they'll eventually converge
    the commit mechanism ensures servers only execute stable entries

extra criterion: leader cannot simply replicate and commit old term's entries
  [Figure 8 example]
  S1 fails to replicate term 2 entry to majority, then fails
  S5 becomes leader in term 3, adds entry, but fails to replicate it
  S1 comes back, becomes leader again
    works on replicating old entry from term 2 to force followers to adopt its log
    are we allowed to commit once the term 2 entry is on a majority of servers?
  Turns out the answer is no! Consider what would happen if we did:
    term 2 entry replicated to S3
    S1 commits it, since it's on a majority of servers
    S1 then fails again
    S5 gets elected for term 4, since it has a term 3 entry at the end of the log
    everybody with a term 2 entry at the end of the log votes for S5
  S5 becomes leader, and now forces *its* log (with the term 3 entry) on others
    term 2 entry at index 2 will get overwritten by term 3 entry
    but it's supposed to be committed!
    therefore, contradicts Leader Completeness property
  solution: wait until S1 has also replicated and committed a term 4 entry
    ensures that S5 can no longer be elected if S1 fails
    thus it's now okay to commit term 2 entry as well

when is it legal for Raft to overwrite log entries? (cf. Figure 7 question)
  they must be uncommitted
  may truncate and overwrite a much longer log
    Figure 7 (f) is a case in point
  e.g., a leader adds many entries to its log, but fails to replicate them
    perhaps it's in a network partition
  other leaders in later terms add entries at the same indices (Fig 7 (a)-(e))
    and commit at least some of them
    now cannot change this log index any more
  outdated server receives AppendEntries, overwrites uncommitted log entries
    even if the log is much longer than the current leader's!
  this is okay, because the leader only responds to clients after entries commit
    so leader who produced the overwritten entries in (f) cannot have done so

*** addendum: lab 2 Raft interface

```
rf.Start(command) (index, term, isleader)
  Lab 3 k/v server's Put()/Get() RPC handlers call Start()
  start Raft agreement on a new log entry
  Start() returns immediately -- RPC handler must then wait for commit
  might not succeed if server loses leadership before committing command
  isleader: false if this server isn't the Raft leader, client should try another
  term: currentTerm, to help caller detect if leader is demoted
  index: log entry to watch to see if the command was committed
ApplyMsg, with Index and Command
  Raft sends a message on the "apply channel" for each
  committed log entry. the service then knows to execute the
  command, and the leader uses the ApplyMsg
  to know when/what to reply to a waiting client RPC.
```