6.824 2018 Lecture 6: Raft (2)

```
Recall the big picture:
  key/value service as the example, as in Lab 3
  goal: same client-visible behavior as single non-replicated server
  goal: available despite minority of failed/disconnected servers
  watch out for network partition and split brain!
  [diagram: clients, k/v layer, k/v table, raft layer, raft log]
  [client RPC -> Start() -> majority commit protocol -> applyCh]
  "state machine", application, service

a few reminders:
  leader commits/executes after a majority replies to AppendEntries
  leader tells commit to followers, which execute (== send on applyCh)
  why wait for just a majority? why not wait for all peers?
    availability requires progress even if minority are crashed
  why is a majority sufficient?
    any two majorities overlap
    so successive leaders' majorities overlap at at least one peer
    so next leader is guaranteed to see any log entry committed by previous leader
  it's majority of all peers (dead and alive), not just majority of live peers

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

as long as the leader stays up:
  clients only interact with the leader
  clients aren't affected by follower actions

things only get interesting when changing leaders
  e.g. after the old leader fails
  how to change leaders without clients seeing anomalies?
    stale reads, repeated operations, missing operations, different order, &c

what do we want to ensure?
  if any server executes a given command in a log entry,
    then no server executes something else for that log entry
  (Figure 3's State Machine Safety)
  why? if the servers disagree on the operations, then a
    change of leader might change the client-visible state,
    which violates our goal of mimicing a single server.
  example:
    S1: put(k1,v1) | put(k1,v2) | ...
    S2: put(k1,v1) | put(k2,x)  | ...
    can't allow both to execute their 2nd log entries!

how can logs disagree after a crash?
  a leader crashes before sending last AppendEntries to all
    S1: 3
    S2: 3 3
    S3: 3 3
  worse: logs might have different commands in same entry!
    after a series of leader crashes, e.g.
        10 11 12 13  <- log entry #
    S1:  3
    S2:  3  3  4
    S3:  3  3  5

Raft forces agreement by having followers adopt new leader's log
  example:
  S3 is chosen as new leader for term 6
  S3 sends an AppendEntries with entry 13
     prevLogIndex=12
     prevLogTerm=5
  S2 replies false (AppendEntries step 2)
  S3 decrements nextIndex[S2] to 12
  S3 sends AppendEntries w/ entries 12+13, prevLogIndex=11, prevLogTerm=3
  S2 deletes its entry 12 (AppendEntries step 3)
  similar story for S1, but have to go back one farther

the result of roll-back:
  each live follower deletes tail of log that differs from leader
  then each live follower accepts leader's entries after that point
  now followers' logs are identical to leader's log

Q: why was it OK to forget about S2's index=12 term=4 entry?
```

```
could new leader roll back *committed* entries from end of previous term?
  i.e. could a committed entry be missing from the new leader's log?
  this would be a disaster -- old leader might have already said "yes" to a client
  so: Raft needs to ensure elected leader has all committed log entries

why not elect the server with the longest log as leader?
  example:
    S1: 5 6 7
    S2: 5 8
    S3: 5 8
  first, could this scenario happen? how?
    S1 leader in term 6; crash+reboot; leader in term 7; crash and stay down
      both times it crashed after only appending to its own log
    next term will be 8, since at least one of S2/S3 learned of 7 while voting
    S2 leader in term 8, only S2+S3 alive, then crash
  all peers reboot
  who should be next leader?
    S1 has longest log, but entry 8 could have committed !!!
    so new leader can only be one of S2 or S3
    i.e. the rule cannot be simply "longest log"

end of 5.4.1 explains the "election restriction"
  RequestVote handler only votes for candidate who is "at least as up to date":
    candidate has higher term in last log entry, or
    candidate has same last term and same length or longer log
  so:
    S2 and S3 won't vote for S1
    S2 and S3 will vote for each other
  so only S2 or S3 can be leader, will force S1 to discard 6,7
    ok since 6,7 not on majority -> not committed -> no reply sent to clients
    -> clients will resend commands in 6,7

the point:
  "at least as up to date" rule ensures new leader's log contains
    all potentially committed entries
  so new leader won't roll back any committed operation

The Question (from last lecture)
  figure 7, top server is dead; which can be elected?

depending on who is elected leader in Figure 7, different entries
  will end up committed or discarded
  c's 6 and d's 7,7 may be discarded OR committed
  some will always remain committed: 111445566

how to roll back quickly
  the Figure 2 design backs up one entry per RPC -- slow!
  lab tester may require faster roll-back
  paper outlines a scheme towards end of Section 5.3
    no details; here's my guess; better schemes are possible
  S1: 4 5 5      4 4 4      4
  S2: 4 6 6  or  4 6 6  or  4 6 6
  S3: 4 6 6      4 6 6      4 6 6
  S3 is leader for term 6, S1 comes back to life
  if follower rejects AppendEntries, it includes this in reply:
    the follower's term in the conflicting entry
    the index of follower's first entry with that term
  if leader has log entries with the follower's conflicting term:
    move nextIndex[i] back to leader's last entry for the conflicting term
  else:
    move nextIndex[i] back to follower's first index for the conflicting term

*** topic: persistence (Lab 2C)

what would we like to happen after a server crashes?
  Raft can continue with one missing server
    but we must repair soon to avoid dipping below a majority
  two strategies:
  * replace with a fresh (empty) server
    requires transfer of entire log (or snapshot) to new server (slow)
    we *must* support this, in case failure is permanent
  * or reboot crashed server, re-join with state intact, catch up
    requires state that persists across crashes
    we *must* support this, for simultaneous power failure
  let's talk about the second strategy -- persistence

if a server crashes and restarts, what must Raft remember?
```

```
  Figure 2 lists "persistent state":
    log[], currentTerm, votedFor
  a Raft server can only re-join after restart if these are intact
  thus it must save them to non-volatile storage
    non-volatile = disk, SSD, battery-backed RAM, &c
    save after each change
    before sending any RPC or RPC reply
  why log[]?
    if a server was in leader's majority for committing an entry,
      must remember entry despite reboot, so any future leader is
      guaranteed to see the committed log entry
  why votedFor?
    to prevent a client from voting for one candidate, then reboot,
      then vote for a different candidate in the same (or older!) term
    could lead to two leaders for the same term
  why currentTerm?
    to ensure that term numbers only increase
    to detect RPCs from stale leaders and candidates

some Raft state is volatile
  commitIndex, lastApplied, next/matchIndex[]
  Raft's algorithms reconstruct them from initial values

persistence is often the bottleneck for performance
  a hard disk write takes 10 ms, SSD write takes 0.1 ms
  so persistence limits us to 100 to 10,000 ops/second
  (the other potential bottleneck is RPC, which takes << 1 ms on a LAN)
  lots of tricks to cope with slowness of persistence:
    batch many new log entries per disk write
    persist to battery-backed RAM, not disk

how does the service (e.g. k/v server) recover its state after a crash+reboot?
  easy approach: start with empty state, re-play Raft's entire persisted log
    lastApplied is volatile and starts at zero, so you may need no extra code!
  but re-play will be too slow for a long-lived system
  faster: use Raft snapshot and replay just the tail of the log

*** topic: log compaction and Snapshots (Lab 3B)

problem:
  log will get to be huge -- much larger than state-machine state!
  will take a long time to re-play on reboot or send to a new server

luckily:
  a server doesn't need *both* the complete log *and* the service state
    the executed part of the log is captured in the state
    clients only see the state, not the log
  service state usually much smaller, so let's keep just that

what constrains how a server discards log entries?
  can't forget un-committed entries -- might be part of leader's majority
  can't forget un-executed entries -- not yet reflected in the state
  executed entries might be needed to bring other servers up to date

solution: service periodically creates persistent "snapshot"
  [diagram: service with state, snapshot on disk, raft log, raft persistent]
  copy of entire state-machine state as of execution of a specific log entry
    e.g. k/v table
  service writes snapshot to persistent storage (disk)
  service tells Raft it is snapshotted through some log index
  Raft discards log before that index
  a server can create a snapshot and discard prefix of log at any time
    e.g. when log grows too long

relation of snapshot and log
  snapshot reflects only executed log entries
    and thus only committed entries
  so server will only discard committed prefix of log
    anything not known to be committed will remain in log

so a server's on-disk state consists of:
  service's snapshot up to a certain log entry
  Raft's persisted log w/ following log entries
  the combination is equivalent to the full log

what happens on crash+restart?
  service reads snapshot from disk
  Raft reads persisted log from disk
    sends service entries that are committed but not in snapshot

what if a follower lags and leader has discarded past end of follower's log?
  nextIndex[i] will back up to start of leader's log
  so leader can't repair that follower with AppendEntries RPCs
  thus the InstallSnapshot RPC
  (Q: why not have leader discard only entries that *all* servers have?)

what's in an InstallSnapshot RPC? Figures 12, 13
  term
  lastIncludedIndex
  lastIncludedTerm
  snapshot data

what does a follower do w/ InstallSnapshot?
  reject if term is old (not the current leader)
  reject (ignore) if follower already has last included index/term
    it's an old/delayed RPC
  empty the log, replace with fake "prev" entry
  set lastApplied to lastIncludedIndex
  replace service state (e.g. k/v table) with snapshot contents

note that the state and the operation history are roughly equivalent
  designer can choose which to send
  e.g. last few operations (log entries) for lagging replica,
    but entire state (snapshot) for a replica that has lost its disk.
  still, replica repair can be very expensive, and warrants attention

The Question:
  Could a received InstallSnapshot RPC cause the state machine to go
  backwards in time? That is, could step 8 in Figure 13 cause the state
  machine to be reset so that it reflects fewer executed operations? If
  yes, explain how this could happen. If no, explain why it can't
  happen.

*** topic: configuration change (not needed for the labs)

configuration change (Section 6)
  configuration = set of servers
  sometimes you need to
    move to a new set of servers, or
    increase/decrease the number of servers
  human initiates configuration change, Raft manages it
  we'd like Raft to cope correctly with failure during configuration change
    i.e. clients should not notice (except maybe dip in performance)

why doesn't a straightforward approach work?
  suppose each server has the list of servers in the current config
  change configuration by telling each server the new list
    using some mechanism outside of Raft
  problem: they will learn new configuration at different times
  example: want to replace S3 with S4
    we get as far as telling S1 and S4 that the new config is 1,2,4
    S1: 1,2,3  1,2,4
    S2: 1,2,3  1,2,3
    S3: 1,2,3  1,2,3
    S4:        1,2,4
  OOPS! now *two* leaders could be elected!
    S2 and S3 could elect S2
    S1 and S4 could elect S1

Raft configuration change
  idea: "joint consensus" stage that includes *both* old and new configuration
    avoids any time when both old and new can choose leader independently
  system starts with Cold
  system administrator asks the leader to switch to Cnew
  Raft has special configuration log entries (sets of server addresses)
  each server uses the last configuration in its own log
  1. leader commits Cold,new to a majority of both Cold and Cnew
  2. after Cold,new commits, leader commits Cnew to servers in Cnew

what if leader crashes at various points in this process?
  can we have two leaders for the next term?
  if that could happen, each leader must be one of these:
    A. in Cold, but does not have Cold,new in log
    B. in Cold or Cnew, has Cold,new in log
    C. in Cnew, has Cnew in log
  we know we can't have A+A or C+C by the usual rules of leader election
  A+B? no, since B needs majority from Cold as well as Cnew
  A+C? no, since can't proceed to Cnew until Cold,new committed to Cold
  B+B? no, since B needs majority from both Cold and Cnew
  B+C? no, since B needs majority from Cnew as well as Cold

good! Raft can switch to a new set of servers w/o risk of two active leaders

*** topic: performance

Note: many situations don't require high performance.
  key/value store might.
  but GFS or MapReduce master might not.

Most replication systems have similar common-case performance:
  One RPC exchange and one disk write per agreement.
  So Raft is pretty typical for message complexity.

Raft makes a few design choices that sacrifice performance for simplicity:
  Follower rejects out-of-order AppendEntries RPCs.
    Rather than saving for use after hole is filled.
    Might be important if network re-orders packets a lot.
  No provision for batching or pipelining AppendEntries.
  Snapshotting is wasteful for big states.
  A slow leader may hurt Raft, e.g. in geo-replication.

These have a big effect on performance:
  Disk writes for persistence.
  Message/packet/RPC overhead.
  Need to execute logged commands sequentially.
  Fast path for read-only operations.

Papers with more attention to performance:
  Zookeeper/ZAB; Paxos Made Live; Harp
```