```
6.824 2015 Lecture 6: Raft

this lecture
  larger topic is fault tolerance via replicated state machines
  Raft -- a much more complete design than straight Paxos

Russ Cox of Google will talk about Go on Thursday
  submit your questions early, so we can get them to Russ

Raft overview
  clients -> leader -> followers -> logs -> execution

Raft vs Paxos?
  Our use of Paxos:
    agrees separately on each client operation
  Raft:
    agrees on each new leader (and on tail of log)
    agreement not required for most client operations
    Raft is Paxos optimized for log appends (more or less)
  why Raft-style leader?
    no dueling proposers (unless leader fails)
    fewer messages, less complexity (unless leader fails)
    well-defined notion of one log being more complete than another
      simplifies switching leaders (and maybe crash recovery)

what about understandability?
  you must decide for yourself
  straight Paxos is simpler than Raft
  but straight Paxos is too simple for practical replication
    everyone extends it in their own way
    and ends up with something more or less like Raft
  Paxos+log+leader probably not simpler than Raft
    though presumably depends on which Paxos variant you choose

is more direct use of Paxos (like Lab 3) ever a win?
  i.e. is a Raft-style leader ever a bad idea?
  geographically spread peers
  a single leader would be far from some clients
  some peers would be slow to other peers (paxos tolerates lag)

let's start w/ Raft w/ no leader change
  for now, reliable leader
  followers may be slow or unreachable (but they do not lose state)
  what do we want?
    1. tolerate a minority of failed followers
    2. converge on same log`
       since replication requires same order of execution
    3. execute only when entry cannot be lost (committed)
       since cannot easily un-do execution or reply to client
  idea for ensuring identical log:
    leader sends log entry, index, and info about *previous* entry
    client can reject (e.g I don't have previous entry!)
    leader backs up for that follower, sends earlier entries
    -> leader forces followers' logs to be identical to leader's
  idea for execution:
    idea #1 means leader knows follower is identical up to some point
    once a majority are identical up to a point,
      leader sends that out as commit point,
      everyone can execute through that point,
      leader can reply to clients
```

```
what to do if the leader crashes?
  other servers time out (no AppendEntries "heart-beats" for a while)
  choose a new leader!
  Raft divides time into terms
  most terms have a leader

what are the dangers in transition to a new leader?
  two leaders
  no leader
  might forget an executed log entry
  logs might end up different (diverge)

leader election first, then log consistency at term boundary

how to ensure at most one leader in a term?
  (look at Figure 2, RequestVote RPC, and Rules for Servers)
  leader must get votes from a majority of servers
  server can cast only one vote per term
  thus at most one server can think it has won
  why a majority?
    the answer is always the same!
    allows fault tolerance (failure of minority doesn't impede progress)
    prevents split brain (at most one candidate can get a majority)
    ensures overlap (at least one in majority has every previously committed log entry)

could election fail to choose any leader?
  yes!
  >= 3 candidates split the vote evenly
  or even # of live servers, two candidates each get half

what happens after an election in which no-one gets majority?
  timeout, increment term, new election
  higher term takes precedence, candidates for older terms quit
  note: timeout must be longer than it takes to complete election!
  note: this means some terms may have no leader, no log entries

how does Raft reduce chances of election failure due to split vote?
  each server delays a random amount of time before starting candidacy
  why is the random delay useful?
    [diagram of times at which servers' delays expire]
    one will choose lowest random delay
    hopefully enough time to elect before next delay expires

how to choose the random delay range?
  too short: 2nd candidate starts before first finishes
  too long: system sits idle for too long after leader fails
  a rough guide:
    suppose it takes 10ms to complete an unopposed election
    and there are five servers
    we want delays to be separated by (say) 20ms
    so random delay from 0 to 100 ms
    plus a few multiples of leader heartbeat interval

remember this random delay idea!
  it's a classic scheme for decentralized soft election; e.g. ethernet

Raft's elections follow a common pattern: separation of safety from progress
  *hard* mechanisms ensure < 2 leaders in one term
    problem: elections can fail (e.g. 3-way split)
  solution: always safe to start a new election in a new term
```

```
    problem: repeated elections can prevent any work getting done
  solution: *soft* mechanisms reduce probability of wasted elections
    heartbeat from leader (remind servers not to start election)
    timeout period (don't start election too soon)
    random delays (give one leader time to be elected)

what if old leader isn't aware a new one is elected?
  perhaps b/c old leader didn't see election messages
  new leader means a majority of servers have incremented currentTerm
    so old leader (w/ old term) can't get majority for AppendEntries
    though a minority may accept old server's log entries...
    so logs may diverge at end of old term...

now let's switch topics to data handling at term boundaries

what do we want to ensure?
  each server executes the same client cmds, in the same order
  i.e. if any server executes, then no server executes something
    else for that log entry
  as long as single leader, we've already seen it makes logs identical
  what about when leader changes?

what's the danger?
  leader of term 3 crashed while sending AppendEntries
    S1: 3
    S2: 3 3
    S3: 3 3
    S2 and S3 might have executed; does Raft preserve it?
  may be a series of crashes, e.g.
    S1: 3
    S2: 3 3 4
    S3: 3 3 5
  thus diff entries for the same index!

roll-back is a big hammer -- forces leader's log on everyone
  in above examples, whoever is elected imposes log on everyone
  example:
    S3 is chosen as new leader for term 6
    S3 wants to send out a new entry (in term 6)
    AppendEntries says previous entry must have term 5
    S2 replies false (AppendEntries step 2)
    S3 decrements nextIndex[S2]
    AppendEntries for the term=5 op, saying prev has term=3
    S2 deletes op from term 4 (AppendEntries step 3)
    (and S1 rejects b/c it doesn't have anything in that entry)

ok, leader will force its own log on followers
  but that's not enough!
  can roll-back delete an executed entry?

when is a log entry executed?
  when leader advances commitIndex/leaderCommit
  when a majority match the leader up through this point

could new leader roll back executed entries from end of previous term?
  i.e. could an executed entry be missing from the new leader's log?
  Raft needs to ensure new leader's log contains every potentially executed entry
  i.e. must forbid election of server who might be missing an executed entry

what are the election rules?
  Figure 2 says only vote if candidate's log "at least as up to date"
```

```
    So leader will be at least as up to date as a majority

  what does "at least as up to date" mean?
    could it mean log is >= length?
    no; example:
      S1: 5 6 7
      S2: 5 8
      S3: 5 8
    first, could this scenario happen? how?
      S1 leader in epoch 6; crash+reboot; leader in epoch 7; crash and stay down
        both times it crashed after only appending to its own log
      S2 leader in epoch 8, only S2+S3 alive, then crash
    who should be next leader?
      S1 has longest log, but entry 8 is committed !!!
        Raft adopts leader's log, so S1 as leader -> un-commit entry 8
        incorrect since S2 may have replied to client
      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 "at least as up to date" voting rule
    compare last entry
    higher term wins
    if equal terms, longer log wins

  so:
    S1 can't get any vote from S2 or S3, since 7 < 8
    S1 will vote for either S2 or S3, since 8 > 7
    S1's operations from terms 6 and 7 will be discarded!
      ok since no majority -> not executed -> no client reply

  the point:
    "at least as up to date" rule causes new leader to have all executed
      entries in its log
    so new leader won't roll back any executed operation
    similar to Paxos: new round ends up using chosen value (if any) of prev round

  The Question
    figure 7, which of a/d/f could be elected?
    i.e. majority of votes from "less up to date" servers?

  the most subtle thing about Raft (figure 8)
    not 100% true that a log entry on a majority is committed
      i.e. will never be forgotten
    figure 8 describes an exception
    S1: 1 2 4
    S2: 1 2
    S3: 1 2
    S4: 1
    S5: 1 3
    S1 was leader in term 2, sends out two copies of 2
    S5 leader in term 3
    S1 in term 4, sends one more copy of 2 (b/c S3 rejected op 4)
    what if S5 now becomes leader?
      S5 can get a majority (w/o S1)
      S5 will roll back 2 and replace it with 3
    could 2 have executed?
      it is on a majority...
      so could S1 have mentioned it in leaderCommit after majority?
      no! very end of Figure 2 says "log[N].term == currentTerm"
      and S1 was in term 4 when sending 3rd copy of 2
    what's Raft's actual commit point?
```

```
      bottom-right of page 310
      "committed once the leader that created the entry has replicated on majority"
      and commit point of one entry commits all before it
        which is how 2 *could* have committed if S1 hadn't lost leadership

  another topic: configuration change (Section 6)
    configuration = set of servers
    how does Raft change the set of servers?
    e.g. every few years might want to retire some, add some
    or move all at once to an entirely new set of server
    or increase/decrease the number of servers

  how might a *broken* configuration change work?
    each server has the list of servers in the current config
    change configuation by changing lists, one by one
    example: want to replace S3 with S4
      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  1,2,4
    OOPS!
      now *two* disjoint group/leaders can form:
        S2 and S3 (who know nothing of new config)
        S1 and S4
      both can process client requests, so split brain

  Raft configuration change
    idea: "join consensus" stage that includes *both* old and new configuration
    leader of old group logs entry that switches to joint consensus
    during joint consensus, leader separately logs in old and new
      i.e. *two* log and *two* agreements on each log entry
      this will force new servers to catch up
      and force new and old logs to be the same
    after majority of old and new have switched to joint consensus,
      leader logs entry that switches to final configuration
    S1: 1,2,3  1,2,3+1,2,4
    S2: 1,2,3
    S3: 1,2,3
    S4:        1,2,3+1,2,4
    if crash but new leader didn't see the switch to joint consensus,
      then old group will continue, no switch, but that's OK
    if crash and new leader did see the switch to joint consensus,
      it will complete the configuration change

  performance
    no numbers on how fast it can process requests
    what are the bottlenecks likely to be?
    disk:
      need to write disk for client data durability, and for protocol promises
      write per client request? so 100 per second?
      could probably batch and get 10,000 to 100,000
    net: a few message exchanges per client request
      10s of microseconds for local LAN message exchange?
      so 100,000 per second?

  Thursday: Russ Cox of Google on Go

  next week: use of a Raft-like protocol in a complex application
```