2018 Lecture 7: Raft (3) -- Snapshots, Linearizability, Duplicate Detection

this lecture:
  Raft snapshots
  linearizability
  duplicate RPCs
  faster gets

*** Raft 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 entries *can't* a server discard?
  un-executed entries -- not yet reflected in the state
  un-committed entries -- might be part of leader's majority

solution: service periodically creates persistent "snapshot"
  [diagram: service state, snapshot on disk, raft log, raft persistent]
  copy of service 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

what happens on crash+restart?
  service reads snapshot from disk
  Raft reads persisted log from disk
  Raft log may start before the snapshot (but definitely not after)
    Raft will (re-)send committed entries on the applyCh
      since applyIndex starts at zero after reboot
    service will see repeats, must detect repeated index, ignore

what if follower's log ends before leader's log starts?
  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

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

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

philosophical note:
  state is often equivalent to operation history
  you can often choose which one to store or communicate
  we'll see examples of this duality later in the course

practical notes:
  Raft layer and service layer cooperate to save/restore snapshots
  Raft's snapshot scheme is reasonable if the state is small
  for a big DB, e.g. if replicating gigabytes of data, not so good
    slow to create and write to disk
  perhaps service data should live on disk in a B-Tree
    no need to explicitly persist, since on disk already
  dealing with lagging replicas is hard, though
    leader should save the log for a while

    or save identifiers of updated records
  and fresh servers need the whole state

*** linearizability

we need a definition of "correct" for Lab 3 &c
  how should clients expect Put and Get to behave?
  often called a consistency contract
  helps us reason about how to handle complex situations correctly
    e.g. concurrency, replicas, failures, RPC retransmission,
         leader changes, optimizations
  we'll see many consistency definitions in 6.824
    e.g. Spinnaker's timeline consistency

"linearizability" is the most common and intuitive definition
  formalizes behavior expected of a single server

linearizability definition:
  an execution history is linearizable if
    one can find a total order of all operations,
    that matches real-time (for non-overlapping ops), and
    in which each read sees the value from the
    write preceding it in the order.

a history is a record of client operations, each with
  arguments, return value, time of start, time completed

example 1:
  |-Wx1-| |-Wx2-|
    |---Rx2---|
      |-Rx1-|
"Wx1" means "write value 1 to record x"
"Rx1" means "a read of record x yielded value 1"
order: Wx1 Rx1 Wx2 Rx2
  the order obeys value constraints (W -> R)
  the order obeys real-time constraints (Wx1 -> Wx2)
  so the history is linearizable

example 2:
  |-Wx1-| |-Wx2-|
    |--Rx2--|
              |-Rx1-|
Wx2 then Rx2 (value), Rx2 then Rx1 (time), Rx1 then Wx2 (value). but
that's a cycle -- so it cannot be turned into a linear order. so this
is not linearizable.

example 3:
|--Wx0--| |--Wx1--|
            |--Wx2--|
        |-Rx2-| |-Rx1-|
order: Wx0 Wx2 Rx2 Wx1 Rx1
so it's linearizable.
note the service can pick the order for concurrent writes.
  e.g. Raft placing concurrent ops in the log.

example 4:
|--Wx0--| |--Wx1--|
            |--Wx2--|
C1: |-Rx2-| |-Rx1-|
C2: |-Rx1-| |-Rx2-|
we have to be able to fit all operations into a single order
  maybe: Wx2 C1:Rx2 Wx1 C1:Rx1 C2:Rx1
    but where to put C2:Rx2?
      must come after C2:Rx1 in time
      but then it should have read value 1
  no order will work:
    C1's reads require Wx2 before Wx1
    C2's reads require Wx1 before Wx2
    that's a cycle, so there's no order
  not linearizable!
so: all clients must see concurrent writes in the same order

example 5:
ignoring recent writes is not linearizable
Wx1   Rx1
    Wx2
this rules out split brain, and forgetting committed writes

You may find this page useful:
https://www.anishathalye.com/2017/06/04/testing-distributed-systems-for-linearizability/

*** duplicate RPC detection (Lab 3)

What should a client do if a Put or Get RPC times out?
  i.e. Call() returns false
  if server is dead, or request dropped: re-send
  if server executed, but request lost: re-send is dangerous

problem:
  these two cases look the same to the client (no reply)
  if already executed, client still needs the result

idea: duplicate RPC detection
  let's have the k/v service detect duplicate client requests
  client picks an ID for each request, sends in RPC
    same ID in re-sends of same RPC
  k/v service maintains table indexed by ID
  makes an entry for each RPC
    record value after executing
  if 2nd RPC arrives with the same ID, it's a duplicate
    generate reply from the value in the table

design puzzles:
  when (if ever) can we delete table entries?
  if new leader takes over, how does it get the duplicate table?
  if server crashes, how does it restore its table?

idea to keep the duplicate table small
  one table entry per client, rather than one per RPC
  each client has only one RPC outstanding at a time
  each client numbers RPCs sequentially
  when server receives client RPC #10,
    it can forget about client's lower entries
    since this means client won't ever re-send older RPCs

some details:
  each client needs a unique client ID -- perhaps a 64-bit random number
  client sends client ID and seq # in every RPC
    repeats seq # if it re-sends
  duplicate table in k/v service indexed by client ID
    contains just seq #, and value if already executed
  RPC handler first checks table, only Start()s if seq # > table entry
  each log entry must include client ID, seq #
  when operation appears on applyCh
    update the seq # and value in the client's table entry
    wake up the waiting RPC handler (if any)

what if a duplicate request arrives before the original executes?
  could just call Start() (again)
  it will probably appear twice in the log (same client ID, same seq #)
  when cmd appears on applyCh, don't execute if table says already seen

how does a new leader get the duplicate table?
  all replicas should update their duplicate tables as they execute
  so the information is already there if they become leader

if server crashes how does it restore its table?
  if no snapshots, replay of log will populate the table
  if snapshots, snapshot must contain a copy of the table

but wait!
  the k/v server is now returning old values from the duplicate table
  what if the reply value in the table is stale?
  is that OK?

example:
  C1   C2
  --   --
  put(x,10)
              first send of get(x), 10 reply dropped
  put(x,20)
              re-sends get(x), gets 10 from table, not 20

what does linearizabilty say?

```
C1: |-Wx10-|          |-Wx20-|
C2:          |-Rx10-------------|
order: Wx10 Rx10 Wx20
so: returning the remembered value 10 is correct

*** read-only operations (end of Section 8)

Q: does the Raft leader have to commit read-only operations in
   the log before replying? e.g. Get(key)?

that is, could the leader respond immediately to a Get() using
  the current content of its key/value table?

A: no, not with the scheme in Figure 2 or in the labs.
   suppose S1 thinks it is the leader, and receives a Get(k).
   it might have recently lost an election, but not realize,
   due to lost network packets.
   the new leader, say S2, might have processed Put()s for the key,
   so that the value in S1's key/value table is stale.
   serving stale data is not linearizable; it's split-brain.

so: Figure 2 requires Get()s to be committed into the log.
    if the leader is able to commit a Get(), then (at that point
    in the log) it is still the leader. in the case of S1
    above, which unknowingly lost leadership, it won't be
    able to get the majority of positive AppendEntries replies
    required to commit the Get(), so it won't reply to the client.

but: many applications are read-heavy. committing Get()s
  takes time. is there any way to avoid commit
  for read-only operations? this is a huge consideration in
  practical systems.

idea: leases
  modify the Raft protocol as follows
  define a lease period, e.g. 5 seconds
  after each time the leader gets an AppendEntries majority,
    it is entitled to respond to read-only requests for
    a lease period without commiting read-only requests
    to the log, i.e. without sending AppendEntries.
  a new leader cannot execute Put()s until previous lease period
    has expired
  so followers keep track of the last time they responded
    to an AppendEntries, and tell the new leader (in the
    RequestVote reply).
  result: faster read-only operations, still linearizable.

note: for the Labs, you should commit Get()s into the log;
      don't implement leases.

Spinnaker optionally makes reads even faster, sacrificing linearizability
  Spinnaker's "timeline reads" are not required to reflect recent writes
    they are allowed to return an old (though committed) value
  Spinnaker uses this freedom to speed up reads:
    any replica can reply to a read, allowing read load to be parallelized
  but timeline reads are not linearizable:
    replica may not have heard recent writes
    replica may be partitioned from the leader!

in practice, people are often (but not always) willing to live with stale
  data in return for higher performance
```