

## 6.824 2020 Lecture 7: Raft (2)

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

as long as the leader stays up:

- clients only interact with the leader
- clients can't see follower states or logs

things get interesting when changing leaders

- e.g. after the old leader fails

- how to change leaders without anomalies?

- diverging replicas, missing operations, repeated operations, &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 S3 has to back up 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

Q: after S1 crashes in term 7, why won't S2/S3 choose 6 as next term?  
 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 -> reply never sent to clients  
 -> clients will resend the discarded commands

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  
 some will always remain committed: 111445566  
 they \*could\* have been committed + executed + replied to  
 some will certainly be discarded: f's 2 and 3; e's last 4,4  
 c's 6,6 and d's 7,7 may be discarded OR committed

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

| Case 1         | Case 2     | Case 3  |
|----------------|------------|---------|
| S1: 4 5 5      | 4 4 4      | 4       |
| S2: 4 6 6 6 or | 4 6 6 6 or | 4 6 6 6 |

S2 is leader for term 6, S1 comes back to life, S2 sends AE for last 6  
 AE has prevLogTerm=6  
 rejection from S1 includes:  
 XTerm: term in the conflicting entry (if any)  
 XIndex: index of first entry with that term (if any)  
 XLen: log length

Case 1 (leader doesn't have XTerm):  
 nextIndex = XIndex

Case 2 (leader has XTerm):  
 nextIndex = leader's last entry for XTerm

Case 3 (follower's log is too short):  
 nextIndex = XLen

\*\*\* topic: persistence (Lab 2C)

what would we like to happen after a server crashes?

Raft can continue with one missing server  
 but failed server must be repaired 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 -- many points in code

or 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 terms only increase, so each term has at most one leader

to detect RPCs from stale leaders and candidates

some Raft state is volatile

commitIndex, lastApplied, next/matchIndex[]

why is it OK not to save these?

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!

this is what Figure 2 does

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 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 (same in mem and disk)]

copy of service state as of execution of a specific log entry

e.g. k/v table

service writes snapshot to persistent storage (disk)

snapshot includes index of last included log entry

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  
 service tells Raft to set lastApplied to last included index  
 to avoid re-applying already-applied log entries

problem: what if follower's log ends before leader's log starts?  
 because follower was offline and leader discarded early part of 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

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'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 snapshot, since on disk already  
 dealing with lagging replicas is hard, though  
 leader should save the log for a while  
 or remember which parts of state have been updated

\*\*\* 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

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

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 history 1:

```

|-Wx1-| |-Wx2-|
|---Rx2---|
|-Rx1-|

```

"Wx1" means "write value 1 to record x"

"Rx1" means "a read of record x yielded value 1"

draw the constraint arrows:

the order obeys value constraints (W → R)  
 the order obeys real-time constraints (Wx1 → Wx2)

this order satisfies the constraints:

Wx1 Rx1 Wx2 Rx2  
 so the history is linearizable

note: the definition is based on external behavior

so we can apply it without having to know how service works

note: histories explicitly incorporates concurrency in the form of  
 overlapping operations (ops don't occur at a point in time), thus good  
 match for how distributed systems operate.

example history 2:

```

|-Wx1-| |-Wx2-|

```

```

|--Rx2--|
      |--Rx1-|

```

draw the constraint arrows:

```

Wx1 before Wx2 (time)
Wx2 before Rx2 (value)
Rx2 before Rx1 (time)
Rx1 before Wx2 (value)

```

there's a cycle -- so it cannot be turned into a linear order. so this history is not linearizable. (it would be linearizable w/o Rx2, even though Rx1 overlaps with Wx2.)

example history 3:

```

|--Wx0--|   |--Wx1--|
           |--Wx2--|
           |--Rx2-|  |--Rx1-|

```

order: Wx0 Wx2 Rx2 Wx1 Rx1

so the history linearizable.

so:

the service can pick either order for concurrent writes.  
e.g. Raft placing concurrent ops in the log.

example history 4:

```

|--Wx0--|   |--Wx1--|
           |--Wx2--|
C1:      |--Rx2-|  |--Rx1-|
C2:      |--Rx1-|  |--Rx2-|

```

what are the constraints?

```

Wx2 then C1:Rx2 (value)
C1:Rx2 then Wx1 (value)
Wx1 then C2:Rx1 (value)
C2:Rx1 then Wx2 (value)
a cycle! so not linearizable.

```

so:

service can choose either order for concurrent writes  
but all clients must see the writes in the same order  
this is important when we have replicas or caches  
they have to all agree on the order in which operations occur

example history 5:

```

|--Wx1-|
      |--Wx2-|
           |--Rx1-|

```

constraints:

```

Wx2 before Rx1 (time)
Rx1 before Wx2 (value)
(or: time constraints mean only possible order is Wx1 Wx2 Rx1)
there's a cycle; not linearizable

```

so:

reads must return fresh data: stale values aren't linearizable  
even if the reader doesn't know about the write  
the time rule requires reads to yield the latest data  
linearizability forbids many situations:  
split brain (two active leaders)  
forgetting committed writes after a reboot  
reading from lagging replicas

example history 6:

suppose clients re-send requests if they don't get a reply  
in case it was the response that was lost:

leader remembers client requests it has already seen  
if sees duplicate, replies with saved response from first execution  
but this may yield a saved value from long ago -- a stale value!

what does linearizability say?

```

C1: |--Wx3-|           |--Wx4-|
C2:      |--Rx3-----|

```

order: Wx3 Rx3 Wx4

so: returning the old saved value 3 is correct

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 linearizability 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 committing 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.

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