Raft (2) FAQ

Q: What are some uses of Raft besides GFS master replication?

A: You could (and will) build a fault-tolerant key/value database using Raft.

You could make the MapReduce master fault-tolerant with Raft.

You could build a fault-tolerant locking service.

Q: When raft receives a read request does it still commit a no-op?

A: Section 8 mentions two different approaches. The leader could send out a heartbeat first; or there could be a convention that the leader can't change for a known period of time after each heartbeat (the lease). Real systems usually use a lease, since it requires less communication.

Section 8 says the leader sends out a no-op only at the very beginning of its term.

Q: The paper states that no log writes are required on a read, but then immediately goes on to introduce committing a no-op as a technique to get the committed. Is this a contradiction or are no-ops not considered log 'writes'?

A: The no-op only happens at the start of the term, not for each read.

Q: I find the line about the leader needing to commit a no-op entry in order to know which entries are committed pretty confusing. Why does it need to do this?

A: The problem situation is shown in Figure 8, where if S1 becomes leader after (b), it cannot know if its last log entry (2) is committed or not. The situation in which the last log entry will turn out not to be committed is if S1 immediately fails, and S5 is the next leader; in that case S5 will force all peers (including S1) to have logs identical to S5's log, which does not include entry 2.

But suppose S1 manages to commit a new entry during its term (term 4). If S5 sees the new entry, S5 will erase 3 from its log and accept 2 in its place. If S5 does not see the new entry, S5 cannot be the next leader if S1 fails, because it will fail the Election Restriction. Either way, once S1 has committed a new entry for its term, it can correctly conclude that every preceding entry in its log is committed.

The no-op text at the end of Section 8 is talking about an optimization in which the leader executes and answers read-only commands (e.g. get("k1")) without committing those commands in the log. For example, for get("k1"), the leader just looks up "k1" in its key/value table and sends the result back to the client. If the leader has just started, it may have at the end of its log a put("k1", "v99"). Should the leader send "v99" back to the client, or the value in the leader's key/value table? At first, the leader doesn't know whether that v99 log entry is committed (and must be returned to the client) or not committed (and must not be sent back). So (if you are using this optimization) a new Raft leader first tries to commit a no-op to the log; if the commit succeeds (i.e. the leader doesn't crash), then the leader knows everything before that point is committed.

Q: How does using the heartbeat mechanism to provide leases (for read-only) operations work, and why does this require timing for

safety (e.g. bounded clock skew)?

A: I don't know exactly what the authors had in mind. Perhaps every AppendEntries RPC the leader sends out says or implies that the no other leader is allowed to be elected for the next 100 milliseconds. If the leader gets positive responses from a majority, then the leader can serve read-only requests for the next 100 milliseconds without further communication with the followers.

This requires the servers to have the same definition of what 100 milliseconds means, i.e. they must have clocks that tick at the close to the same rate.

Q: What exactly do the C_old and C_new variables in Section 6 (and Figure 11) represent? Are they the leader in each configuration?

A: They are the set of servers in the old/new configuration.

The paper doesn't provide details. I believe it's the identities (network names or addresses) of the servers.

Q: When transitioning from cluster C_old to cluster C_new, how can we create a hybrid cluster C_{old,new}? I don't really understand what that means. Isn't it following either the network configuration of C_old or of C_new? What if the two networks disagreed on a connection?

A: During the period of joint consensus (while Cold,new is active), the leader is required to get a majority from both the servers in Cold and the servers in Cnew.

There can't really be disagreement, because after Cold,new is committed into the logs of both Cold and Cnew (i.e. after the period of joint consensus has started), any new leader in either Cold or Cnew is guaranteed to see the log entry for Cold,Cnew.

Q: I'm confused about Figure 11 in the paper. I'm unsure about how exactly the transition from 'C_old' to 'C_old,new' to 'C_new' goes. Why is there the issue of the cluster leader not being a part of the new configuration, where the leader steps down once it has committed the 'C_new' log entry? (The second issue mentioned in Section 6)

A: Suppose C_old={S1,S2,S3} and C_new={S4,S5,S6}, and that S1 is the leader at the start of the configuration change. At the end of the configuration change, after S1 has committed C_new, S1 should not be participating any more, since S1 isn't in C_new. One of S4, S5, or S6 should take over as leader.

Q: About cluster configuration: During the configuration change time, if we have to stop receiving requests from the clients, then what's the point of having this automated configuration step? Doesn't it suffice to just 1) stop receiving requests 2) change the configurations 3) restart the system and continue?

A: The challenge here is ensuring that the system is correct even if there are failures during this process, and even if not all servers get the "stop receiving requests" and "change the configuration" commands at the same time. Any scheme has to cope with the possibility of a mix of servers that have and have not seen or completed the configuration change -- this is true even of a non-automated system. The paper's protocol is one way to solve this problem.

Q: The last two paragraphs of section 6 discuss removed servers interfering with the cluster by trying to get elected even though

theyâ€™ve been removed from the configuration.
Wouldnâ€™t a simpler solution be to require servers to be
shut down when they leave the configuration? It seems that leaving the
cluster implies that a server canâ€™t send or receive RPCs to
the rest of the cluster anymore, but the paper doesnâ€™t
assume that. Why not? Why canâ€™t you assume that the servers
will shut down right away?

A: I think the immediate problem is that the Section 6 protocol doesn't
commit Cnew to the old servers, it only commits Cnew to the servers in
Cnew. So the servers that are not in Cnew never learn when Cnew takes
over from Cold,new.

The paper does say this:

> When the new configuration has been committed under the rules of Cnew,
> the old configuration is irrelevant and servers not in the new
> configuration can be shut down.

So perhaps the problem only exists during the period of time between the
configuration change and when an administrator shuts down the old
servers. I don't know why they don't have a more automated scheme.

Q: How common is it to get a majority from both the old and new
configurations when doing things like elections and entry commitment,
if it's uncommon, how badly would this affect performance?

A: I imagine that in most cases there is no failure, and the leader gets
both majorities right away. Configuration change probably only takes a
few round trip times, i.e. a few dozen milliseconds, so the requirement
to get both majorities will slow the system down for only a small amount
of time. Configuration change is likely to be uncommon (perhaps every
few months); a few milliseconds of delay every few months doesn't seem
like a high price.

Q: And how important is the decision to have both majorities?

A: The requirement for both majorities is required for correctness, to
cover the possibility that the leader fails during the configuration
change.

Q: Just to be clear, the process of having new members join as non-voting
entities isn't to speed up the process of replicating the log, but
rather to influence the election process? How does this increase
availability? These servers that need to catch up are not going to be
available regardless, right?

A: The purpose of non-voting servers is to allow those servers to get a
complete copy of the leader's log without holding up new commits. The
point is to allow a subsequent configuration change to be quick. If the
new servers didn't already have nearly-complete logs, then the leader
wouldn't be able to commit Cold,new until they caught up; and no new
client commands can be executed between the time Cold,new is first sent
out and the time at which it is committed.

Q: If the cluster leader does not have the new configuration, why doesn't
it just remove itself from majority while committing C_new, and then
when done return to being leader? Is there a need for a new election
process?

A: Is this about the "second issue" in Section 6? The situation they describe is one in which the leader isn't in the new configuration at all. So after Cnew is committed, the leader shouldn't be participating in Raft at all.

Q: How does the non-voting membership status work in the configuration change portion of Raft. Does that server state only last during the changeover (i.e. while c_new not committed) or do servers only get full voting privileges after being fully "caught up"? If so, at what point are they considered "caught up"?

A: The paper doesn't have much detail here. I imagine that the leader won't start the configuration change until the servers to be added (the non-voting members) are close to completely caught up. When the leader sends out the Cold,new log entry in AppendEntries RPCs to those new servers, the leader will bring them fully up to date (using the Figure 2 machinery). The leader won't be able to commit the Cold,new message until a majority of those new servers are fully caught up. Once the Cold,new message is committed, those new servers can vote.

Q: I don't disagree that having servers deny RequestVotes that are less than the minimum election timeout from the last heartbeat is a good idea (it helps prevent unnecessary elections in general), but why did they choose that method specifically to prevent servers not in a configuration running for election? It seems like it would make more sense to check if a given server is in the current configuration. E.g., in the lab code we are using, each server has the RPC addresses of all the servers (in the current configuration?), and so should be able to check if a requestVote RPC came from a valid (in-configuration) server, no?

Q: I agree that the paper's design seems a little awkward, and I don't know why they designed it that way. Your idea seems like a reasonable starting point. One complication is that there may be situations in which a server in Cnew is leader during the joint consensus phase, but at that time some servers in Cold may not know about the joint consensus phase (i.e. they only know about Cold, not Cold,new); we would not want the latter servers to ignore the legitimate leader.

Q: When exactly does joint consensus begin, and when does it end? Does joint consensus begin at commit time of "C_{o,n}"?

A: Joint consensus is in progress when the current leader is aware of Cold,new. If the leader doesn't manage to commit Cold,new, and crashes, and the new leader doesn't have Cold,new in its log, then joint consensus ends early. If a leader manages to commit Cold,new, then joint consensus has not just started but will eventually complete, when a leader commits Cnew.

Q: Can the configuration log entry be overwritten by a subsequent leader (assuming that the log entry has not been committed)?

A: Yes, that is possible, if the original leader trying to send out Cold,new crashes before it commits the Cold,new.

Q: How can the "C_{o,n}" log entry ever be committed? It seems like it must be replicated to a majority of "old" servers (as well as the "new" servers), but the append of "C_{o,n}" immediately transitions the old server to new, right?

A: The commit does not change the set of servers in Cold or Cnew. For example, perhaps the original configuration contains servers S1, S2, S3; then Cold is {S1,S2,S3}. Perhaps the desired configuration is S4, S5, S6; then

Cnew is {S4,S5,S6}. Once Cnew is committed to the log, the configuration is Cnew={S4,S5,S6}; S1,S2, and S3 are no longer part of the configuration.

Q: When snapshots are created, is the data and state used the one for the client application? If it's the client's data then is this something that the client itself would need to support in addition to the modifications mentioned in the raft paper?

A: Example: if you are building a key/value server that uses Raft for replication, then there will be a key/value module in the server that stores a table of keys and values. It is that table of keys and values that is saved in the snapshot.

Q: The paper says that "if the follower receives a snapshot that describves a prefix of its log, then log entries covered by the snapshot are deleted but entries following the snapshot are retained". This means that we could potentially be deleting operations on the state machinne.

A: I don't think information will be lost. If the snapshot covers a prefix of the log, that means the snapshot includes the effects of all the operations in that prefix. So it's OK to discard that prefix.

Q: It seems that snapshots are useful when they are a lot smaller than applying the sequence of updates (e.g., frequent updates to a few keys). What happens when a snapshot is as big as the sum of its updates (e.g., each update inserts a new unique key)? Are there any cost savings from doing snapshots at all in this case?

A: If the snapshot is about as big as the log, then there may not be a lot of value in having snapshots. On the other hand, perhaps the snapshot organizes the data in a way that's easier to access than a log, e.g. in a sorted table. Then it might be faster to re-start the service after a crash+reboot from a snapshotted table than from the log (which you would have to sort).

It's much more typical, however, for the log to be much bigger than the state.

Q: Also wouldn't a InstallSnapshot incur heavy bandwidth costs?

A: Yes, if the state is large (as it would be for e.g. a database). However, this is not an easy problem to solve. You'd probably want the leader to keep enough of its log to cover all common cases of followers lagging or being temporarily offline. You might also want a way to transfer just the differences in server state, e.g. just the parts of the database that have changed recently.

Q: Is there a concern that writing the snapshot can take longer than the election timeout because of the amount of data that needs to be appended to the log?

A: You're right that it's a potential problem for a large server. For example if you're replicating a database with a gigabyte of data, and your disk can only write at 100 megabytes per second, writing the snapshot will take ten seconds. One possibility is to write the snapshot in the background (i.e. arrange to not wait for the write, perhaps by doing the write from a child process), and to make sure that snapshots are created less often than once per ten seconds.

Q: Under what circumstances would a follower receive a snapshot that is a prefix of its own log?

A: The network can deliver messages out of order, and the RPC handling system can execute them out of order. So for example if the leader sends a snapshot for log index 100, and then one for log index 110, but the network delivers the second one first.

Q: Additionally, if the follower receives a snapshot that is a prefix of its log, and then replaces the entries in its log up to that point, the entries after that point are ones that the leader is not aware of, right?

A: The follower might have log entries that are not in a received snapshot if the network delays delivery of the snapshot, or if the leader has sent out log entires but not yet committed them.

Q: Will those entries ever get committed?

A: They could get committed.

Q: How does the processing of InstallSnapshot RPC handle reordering, when the check at step 6 references log entries that have been compacted? Specifically, shouldn't Figure 13 include: 1.5: If lastIncludedIndex < commitIndex, return immediately. or alternatively 1.5: If there is already a snapshot and lastIncludedIndex < currentSnapshot.lastIncludedIndex, return immediately.

A: I agree -- for Lab 3B the InstallSnapshot RPC handler must reject stale snapshots. I don't know why Figure 13 doesn't include this test; perhaps the authors' RPC system is better about order than ours. Or perhaps the authors intend that we generalize step 6 in Figure 13 to cover this case.

Q: What happens when the leader sends me an InstallSnapshot command that is for a prefix of my log, but I've already undergone log compaction and my snapshot is ahead? Is it safe to assume that my snapshot that is further forward subsumes the smaller snapshot?

A: Yes, it is correct for the recipient to ignore an InstallSnapshot if the recipient is already ahead of that snapshot. This case can arise in Lab 3, for example if the RPC system delivers RPCs out of order.

Q: How do leaders decide which servers are lagging and need to be sent a snapshot to install?

A: If a follower rejects an AppendEntries RPC for log index i1 due to rule #2 or #3 (under AppendEntries RPC in Figure 2), and the leader has discarded its log before i1, then the leader will send an InstallSnapshot rather than backing up nextIndex[].

Q: In actual practical use of raft, how often are snapshots sent?

A: I have not seen an analysis of real-life Raft use. I imagine people using Raft would tune it so that snapshots were rarely needed (e.g. by having leaders keep lots of log entries with which to update lagging followers).

Q: Is InstallSnapshot atomic? If a server crashes after partially installing a snapshot, and the leader re-sends the InstallSnapshot RPC, is this idempotent like RequestVote and AppendEntries RPCs?

A: The implementation of InstallSnapshot must be atomic.

It's harmless for the leader to re-send a snapshot.

Q: Why is an offset needed to index into the data[] of an InstallSNapshot RPC, is there data not related to the snapshot? Or does it overlap previous/future chunks of the same snapshot? Thanks!

A: The complete snapshot may be sent in multiple RPCs, each containing a different part ("chunk") of the complete snapshot. The offset field indicates where this RPC's data should go in the complete snapshot.

Q: How does copy-on-write help with the performance issue of creating snapshots?

A: The basic idea is for th server to fork(), giving the child a complete copy of the in-memory state. If fork() really copied all the memory, and the state was large, this would be slow. But most operating systems don't copy all the memory in fork(); instead they mark the pages as "copy-on-write", and make them read-only in both parent and child. Then the operating system will see a page fault if either tries to write a page, and the operating system will only copy the page at that point. The net effect is usually that the child sees a copy of its parent process' memory at the time of the fork(), but with relatively little copying.

Q: What data compression schemes, such as VIZ, ZIP, Huffman encoding, etc. are most efficient for Raft snapshotting?

A: It depends on what data the service stores. If it stores images, for example, then maybe you'd want to compress them with JPEG.

If you are thinking that each snapshot probably shares a lot of content with previous snapshots, then perhaps you'd want to use some kind of tree structure which can share nodes across versions.

Q: Does adding an entry to the log count as an executed operation?

A: No. A server should only execute an operation in a log entry after the leader has indicated that the log entry is committed. "Execute" means handing the operation to the service that's using Raft. In Lab 3, "execute" means that Raft gives the committed log entry to your key/value software, which applies the Put(key,value) or Get(key) to its table of key/value pairs.

Q: According to the paper, a server disregards RequestVoteRPCs when they think a current leader exists, but then the moment they think a current leader doesn't exist, I thought they try to start their own election. So in what case would they actually cast a vote for another server?

For the second question, I'm still confused: what does the paper mean when it says a server should disregard a RequestVoteRPC when it thinks a current leader exists at the end of Section 6? In what case would a server think a current leader doesn't exist but hasn't started its own election? Is it if the server thinks it hasn't yet gotten a heartbeat from the server but before its election timeout?

A: Each server waits for a randomly chosen election timeout; if it hasn't heard from the leader for that whole period, and no other server has started an

election, then the server starts an election. Whichever server's election timer expires first is likely to get votes from most or all of the servers before any other server's timer expires, and thus is likely to win the election.

Suppose the heartbeat interval is 10 milliseconds (ms). The leader sends out heartbeats at times 10, 20, and 30.

Suppose server S1 doesn't hear the heartbeat at time 30. S1's election timer goes off at time 35, and S1 sends out RequestVote RPCs.

Suppose server S2 does hear the heartbeat at time 30, so it knows the server was alive at that time. S2 will set its election timer to go off no sooner than time 40, since only a missing heartbeat indicates a possibly dead server, and the next heartbeat won't come until time 40. When S2 hears S1'a RequestVote at time 35, S2 can ignore the RequestVote, because S2 knows that it heard a heartbeat less than one heartbeat interval ago.

Q: I'm a little confused by the "how to roll back quickly" part of the Lecture 6 notes (and the corresponding part of the paper):

```
  paper outlines a scheme towards end of Section 5.3:
  if follower rejects, includes this in reply:
    the term of the conflicting entry
    the index of the first entry for conflicting term
  if leader knows about the conflicting term:
    move nextIndex[i] back to its last entry for the conflicting term
  else:
    move nextIndex[i] back to follower's first index
```

I think according to the paper, the leader should move nextIndex[i] to the index of the first entry for conflicting term. What does the situation "if leader knows about the conflicting term" mean?

A: The paper's description of the algorithm is not complete, so we have to invent the details for ourselves. The notes have the version I invented; I don't know if it's what the authors had in mind.

The specific problem with the paper's "index of the first entry for the conflicting term" is that the leader might no have entries at all for the conflicting term. Thus my notes cover two cases -- if the server knows about the conflicting term, and if it doesn't.

Q: What are the tradeoffs in network/ performance in decreasing nextIndex by a factor of 2 each time at each mismatch? i.e. first by 1,2,4, 8 and so on

A: The leader will overshoot by up to a factor of two, and thus have to send more entries than needed. Of course the Figure 2 approach is also wasteful if one has to back up a lot. Best might be to implement something more precise, for example the optimization outlined towards the end of section 5.3.

Q: Unrelatedly - How does your experience teaching Raft and Paxos correspond to section 9.1 of the paper? Do your experiences support their findings?

A: I was pretty happy with the 6.824 Paxos labs from a few years ago. I'm pretty happy with the current Raft labs too. The Raft labs are more ambitious: unlike the Paxos labs, the Raft labs have a leader, persistence, and snapshots. I don't think we have any light to shed on

the findings in Section 9.1; we didn't perform a side-by-side
experiment on the students, and our Raft labs are noticeably more
ambitious.

Q: What has been the impact of Raft, from the perspective of academic
researchers in the field? Is it considered significant, inspiring,
non-incremental work? Or is it more of "okay, this seems like a
natural progression, and is a bit easier to teach, so let's teach
this?"

A: The Raft paper does a better job than any paper I know of in explaining
modern replicated state machine techniques. I think it has inspired lots
of people to build their own replication implementations.

Q: The paper states that there are a fair amount of implementations of Raft out
in the wild. Have there been any improvement suggestions that would make sense
to include in a revised version of the algorithm?

A: Here's an example:

https://www.cl.cam.ac.uk/~ms705/pub/papers/2015-osr-raft.pdf