Linearizable Raft Quorum Reads

12/18/2020

Bernard Zhao <bernardzhao@berkeley.edu>

Richard Huang <richardh@berkeley.edu>

Aditya Srivastava <aditya.srivastava@berkeley.edu>

The problem

In a distributed consensus algorithm like Raft, the leader in charge of coordination is the primary bottleneck for operations for clients. Can we take a common fast path like read operations and make it faster by avoiding going through the leader directly, without sacrificing consistency? A good number of real workloads are read heavy, which means that speeding this up can be a big deal.

Existing solutions

This project is heavily inspired by Linearizable Paxos Quorum Reads (LPQR) [1], a Paxos variant that achieves quorum reads on Paxos. Raft is equally if not more popular, can we get the same gains here? Other consensus algorithms have tried to perform scalable reads or quorum reads, but these optimizations have yet to be attempted with Raft and done in a linearizable way. The closest existing implementation is quorum reads in Raft on top of CockroachDB, which is only serializable using HLC timestamps [2].

A better comparison can be made to how Raft reads are commonly performed:

In the classical case, a read is just like a write or any operation on the distributed log. In Raft, a client would send a read to the leader, it would get appended to the leader's log, and sent to followers to be replicated. Once a majority is reached, the leader can commit the entry to it's log, and then the client can be served the read of data. The entry in this case doesn't contain any information, it serves as a way to perform a consistent read.

So how can reads in Raft be performed linearizably but not in the naive way? The original (extended) Raft paper [3] has a suggestion:

> *Read-only operations can be handled without writing anything into the log. However, with no additional measures, this would run the risk of returning stale data, since the leader responding to the request might have been superseded by a newer leader of which it is unaware. Linearizable reads must not return stale data, and Raft needs two extra precautions to guarantee this without using the log.*
>
> *First, a leader must have the latest information on which entries are committed. The Leader Completeness Property guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank no-op entry into the log at the start of its term.*
>
> *Second, a leader must check whether it has been deposed before processing a read-only request (its information may be stale if a more recent leader has been elected). Raft handles this by having the leader exchange heart-beat messages with a majority of the cluster before responding to read-only requests.*
>
> *Alternatively, the leader could rely on the heartbeat mechanism to provide a form of lease [9], but this would rely on timing for safety (it assumes bounded clock skew).*

To further reduce latency from round trip heartbeat messages, timing can be used to only need to contact followers once, as suggested in the last line. Using a "leader lease" based on timing means a leader is only valid as long as its lease is active. This should expire leaders without communication but relies on bounded clock drift.

Avoiding complicated leasing and timing that requires clocks, LRQR gives us a new linearizable way to perform reads that should achieve better cluster utilization and scalability by bypassing the leader.

Approach

The quorum reads approach has already proven to be possible in Paxos [1], a similar consensus algorithm. Can the same technique be applied to Raft or are there subtleties that prevent it?

The fundamental intuition is that if Raft and Paxos are so similar, implementing Linearizable Quorum Reads should also be a similar process in Raft. To solve the problem we will attempt to produce both an implementation of Linearizable Quorum Reads on Raft and vanilla Raft to compare it to. We chose to use Michael's existing Frankenpaxos framework to take advantage of its existing visualizations [4].

The protocol

We chose to adopt the same two phases used in a read as described in LPQR, but with its own Raft specific quirks.

Quorum Read Phase The client sends a read command to a quorum of participants (e.g. a majority), and receives the latest entry appended to its log, the latest committed entry, and the output of running the command against its state machine. The client will make sure to record those values for the response with the latest entry between all of the received responses, and this includes the output that will be returned as a value of this read.

Rinse Phase To make sure that quorum read value is committed, the client will have to check that that entry is committed eventually. This can be done on any individual participant, because the commit is a global message that the leader delivers, ensuring that if the entry is committed on any participant, it must have already been committed on the leader. When this happens, the client can finish the read with the output at the original latest accepted entry, or else try again by repeating this process.

Optimizations This process normally takes at least two round trips of messages between a client and a quorum of participants, but we can avoid the rinse phase if specific conditions are true. If the latest accepted slot within the quorum has been accepted, we can skip the rinse and read directly. This is the case where the system is already in consensus, because for this to be possible no new commands are being written and all existing commands have been committed as well.

Why this works At a high level, the main idea behind this setup is that by asking the quorum and recording the latest entry index we make sure that we perform a read that is up to date and comes after any other write. Then we need to wait until the value is committed, as in the write actually happens, and then we can return the read. This way we can be sure that this read comes after that write in the order of operations, and recording the output at that index also ensures that it isn't affected by writes that follow it.

Let's think about these possible cases:

- A write followed by a read
  - For the write to have happened prior, this means that the Raft leader must have committed the entry to it's log. That means that the entry appears on a write quorum of the followers. For the quorum read to fail and miss this write, it would need to have not seen that entry as the latest entry in its quorum, which is impossible since it will always intersect

with that quorum. Therefore, once any write is committed, every read after should reflect that write.

- A read followed by a write
  - If the write follows the read, that means that the read entry index i comes before the new written entry index j. Once entry index i is committed the read will eventually finish, on a state machine with actions up until entry i. Therefore no read should ever reflect a write that follows i.

Flexible Quorums Flexible quorums are also possible for Raft and they can be leveraged to further improve its performance. As long as the read and write quorums intersect, or in the case of Raft the election and commit quorums, the algorithm still works correctly but will need to contact more or less participants to perform a write or read. For example, we can increase the commit quorum and require that more participants append an entry before we consider it committed, which also reduces the number machines a client will need to contact to perform a read quorum. This greatly affects the number of requests quorum readers will need to send and can be leveraged to improve performance in differing workloads.

In our evaluation, we characterize the performance of LRQR under the simple majority quorum approach as well as several different grid quorum configurations. A grid quorum is a generalization of the quorum concept in which quorum participants occupy a position on an m x n grid, where each row represents a read quorum and each column represents a write quorum. It follows that to make sure a write is committed, an entry must be committed across m quorum participants constituting a write quorum column. Likewise for reads, n quorum participants constituting a read quorum must be queried before returning a result successfully. In a read or write heavy workload, flexible quorums such as a grid quorum can be utilized to more effectively tune performance of the system.

Visual Demos

You can visualize and play around with the implementations yourself here:

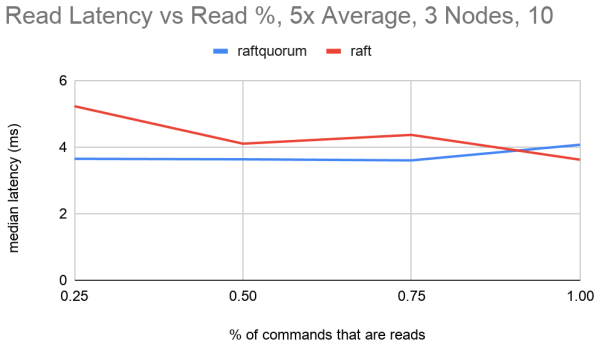
- Normal Raft (with heartbeat reads)
- Quorum Read Raft

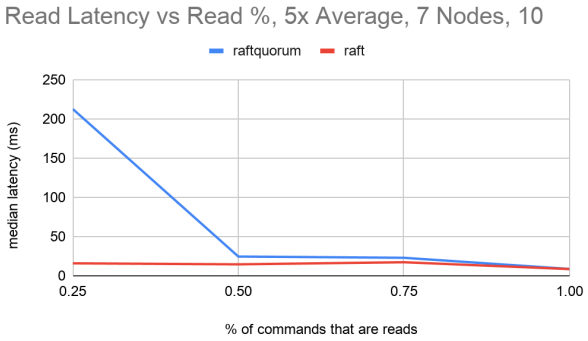
All of the code can be found here

Evaluation

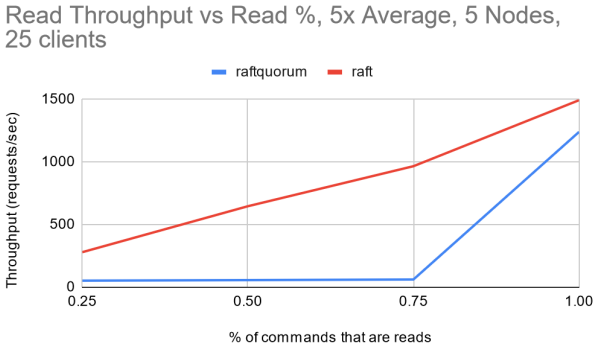
We used Frankenpaxos' benchmarking framework to test out the performance and behavior of our implementation. These ran on top of AWS instances in us west north california, t2.xlarge with 4vCPU 16GiB.

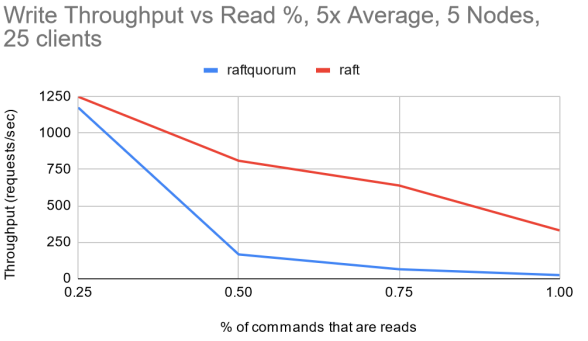
We haven't had the time to exhaust all possible tests, but we made sure to test different Raft cluster sizes, differing amounts of reads and writes in workloads, and differing client load.





We found that raft quorum median read latency is not that different from normal raft, unless we run workloads that are mostly writes. As more writes are happening at once and the latest index is much farther ahead of the committed index, the time it takes for a read to complete suffers heavily. This is mainly due to a few outliers, the max raft quorum read latency is an order of magnitude greater than of normal raft. This is likely due to repeated rinses, of which are repeated in 100ms intervals until the read is completed. Reducing this value may help the performance here. As a result both the read and write throughput are heavily affected, because the clients are stalled on sending new commands.

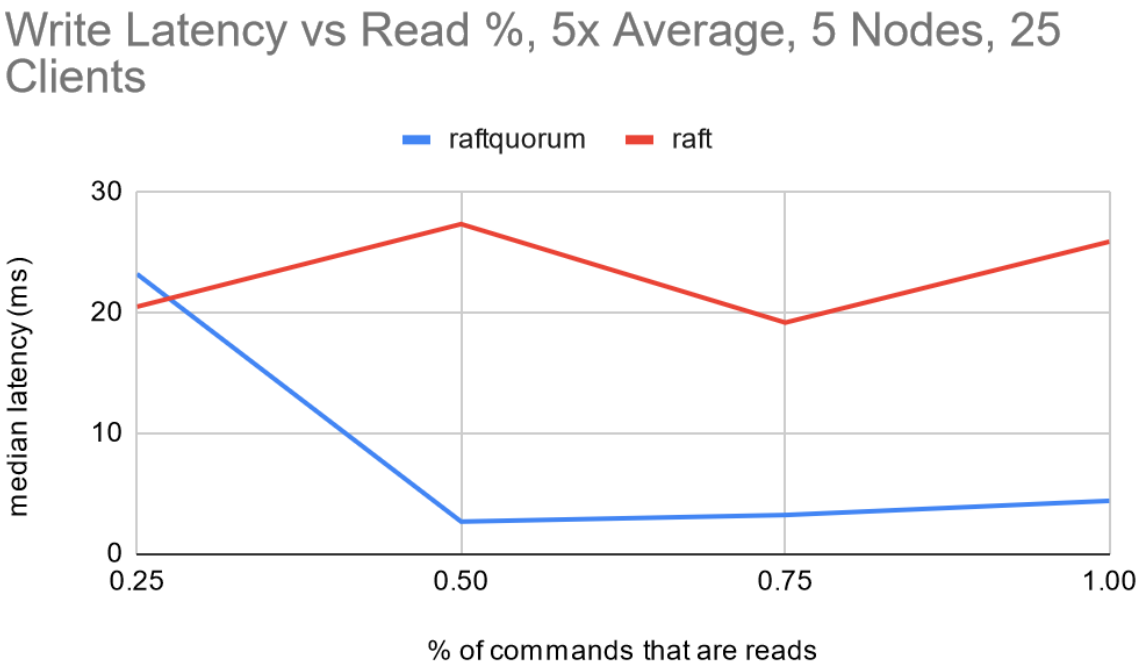




This actually highlights a limitation of the framework: read and write throughput is measured together over time intervals.To get clearer results about how each type of command is affected in the future we should adopt a secondary approach of designating specific client readers and writers.

Interestingly enough, we also found that write latency actually was actually reduced in raft quorums when compared to raft, which is a good sign that the leader was less bottlenecked from quorum reads and was able to service writes

faster.



These results indicate that there is still a lot of possible improvement and promise that quorum reads are useful. To even further speed up rinses, we could even try sending rinses to the leader with no delay to try to get responses ASAP, but as a trade off the leader would begin to see more messages again when consensus has not been reached.

Testing Grid Quorums also yielded results that showed differing tradeoffs as expected, depending on the read and write rows and columns set up prior. It would likely perform better as we scale up the number of participant nodes as the read and write roles are spread out more between participants.

You can view all of the experimental results, data, and charts here.

Future work

- Benchmark and test while tweaking even more different parameters
  - Further explore the space of possible quorums we could use
  - More fully characterize the configurations under variable conditions such as number of clients interacting with the system and read/write ratios.

  - Experiment with different rinse time intervals / schemes like exponential backoff, rinse to the leader if possible, etc.
- Port features onto a popular Raft implementation
- Prove that the LRQR algorithm we use is correct.

Bibliography

1. Charapko, Aleksey, Ailidani Ailijiang, and Murat Demirbas. "Linearizable quorum reads in Paxos." *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. 2019.
2. Arora, Vaibhav, et al. "Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols." *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. 2017.
3. Ongaro, Diego. *Consensus: Bridging theory and practice*. Diss. Stanford University, 2014.
4. https://github.com/mwhittaker/frankenpaxos