Flexible Chained Raft

Alec Diraimondo, Advai Podduturi
University of Illinois at Urbana-Champaign
{alecjd2,advairp2}@illinois.edu

Abstract

Distributed consensus is required for many modern day applications such as replicated state machines, distributed file systems, and databases. However traditional consensus algorithm research has to make fine-grained assumptions about fault behavior. have been developed in research for simplicity and understandability. We propose that more specialized design decisions can be made to improve the performance of consensus algorithms by expanding the model of fault-tolerances to be more applicable to the faults experienced in practice. In this paper we describe two optimizations of Raft that we call Flexible Chained Raft intended for better performance under specific, understudied fault models such as network partitions, and fail-slow faults.

1 Introduction

Most large scale software companies (hyperscalers) are reliant on consensus algorithms to control at least some part of their critical infrastructure. As a result of this, there is warranted motivation to find ways to adapt traditional consensus algorithms to the changing landscape of networks and systems found in practice. By far the two most common consensus algorithms applied in these domains are Raft[6] and Paxos[4]. In this paper, we try to reason about some variants of the Raft consensus protocol that we deemed favorable over the original protocol for certain fault scenarios that may be encountered in practice such as network partitioning and crash-faults with recovery.

2 Motivation

For hyper-scalar cloud service providers, availability, specifically client facing availability, is the most commonly used metric for evaluating system effectiveness. However, needed qualities such as replication and consistency across large cloud-scale systems requires the use of distributed consensus at some level. As Distributed Partial synchronous consensus is expensive, this is typically achieved by a small consensus cluster of five to seven nodes laying at the bottom of the hierarchacal system structure. Since distributed consensus tolerates $\frac{1}{2}$ crash-faults, this provides a degree of crash-fault tolerance, while minimizing the cost to latency and availability of the dependent services.

Raft was originally proposed in 2014 as an easier to understand and implement consensus algorithm when compared to Paxos. Despite those claims, the two protocols are quite similar fundamentally. Specifically, research has shown that there exists a formal mapping between the two under moderate assumptions (named Raft*) using refinement mapping [7]. The result of such work has critically given the ability to port specific optimizations from Paxos to Raft. As a result of Paxos being an active area of research for over twenty years now, this means that lots of Raft research work can be expedited using these previos findings and ideas. This also allows one to view distributed consensus more holistically, and may change the way we approach research in these areas.

We believe that there is lots of research to be done in many areas of fault-tolerant distributed consensus, but particularly Raft optimizations. Crypto-currencies have revived research in the area of fault-tolerance, specifically byzantine fault tolerance. In more recent years, new byzantine fault tolerant consensus algorithms such as Tendermint, Casper FFG [1], and HotStuff [8] have utilized an immutable chain-based log structure. This is a paradigm shift from the usual mutable log structures that Raft, Paxos, PBFT, and all other traditional consensus algorithms use. Some less formal, raft-specific immutable log variants are also being explored in recent times [2]. We believe that as a result of crypto-currencies being motivated almost solely by practical applications, the research in byzantine fault tolerance is much more in line with practical applications and scenarios.

As a result, we wanted to explore some ideas from recent Paxos and Raft optimizations to try and find improvements under specific use cases. In particular, we were interested in using the idea of immutable log entries mentioned earlier. Immutability allows systems to be more robust to concurrency and other bugs, whilst increasing observability of the systems behavior. As we progress into the next decade, systems will get contine to get larger and bugs harder to find. We believe the fundamental concern of a fault-tolerant consensus algorithm in practice should be its reliability, not performance. As a result, we attempt to show how one can mantain (and even improve) performance such as latency and throughput whilst using an immutable log structure for Raft. We call the variant Flexible Chained Raft, as we also implement flexible quorums from Flexible Paxos [3]. By making advantageous and justifiable memory tradeoffs, we believe the added immutabilty allows distributed consensus to be more understandable both to implement and reason about.

(WHY IMMUTABLE DATA W/E citation)

3 Solution

3.1 Flexible Quorums

One such variation of Paxos that has been proven to be applicable to Raft was Flexible Paxos [3]. The main idea of the Flexible Paxos variant is proving that one can relax the strict majority quorums that are used in both Paxos and Raft. We define a strict majority quorum for crash-faults as $[n/2] + 1$ (where n is the number of nodes in the cluster). Many proofs of distributed consensus rely on the idea of a quorum intersection. Put simply, if we build two quorums to make some decision, it is necessary to prove that at least one non-faulty process must lie in the intersection of the quorums to avoid violations of safety and liveness. It is trivial to see why this would be true in the case of two strict majority quorums where f < n/2, with f being the number of crash-faults tolerated. Flexible Paxos proposes that only the following equation must hold true in order for one to achieve a quorum intersection (and thus keep the safety property of replication/agreement):

$$|Q1| + |Q2| > N.$$

Where $Q1$ refers to phase-1 quorum (leader election), and $Q2$ refers to phase 2 quorum (commit acceptance). In multi-decree Paxos, invoking a commit quorum is much more common than a leader election quorum. Thus, the paper claims that by reducing the quorum needed to commit values and increasing the leader election quorum size, we can achieve higher throughput and lower latency in practice (assuming normal network conditions). One of our contributions for this project was to apply flexible quorums to Raft. However to do so it is important to note the subtle differences between the safety conditions that both protocols require. The safety proof for Raft relies on the idea of a single leader at all times. This means that there is no way we can guarantee that Raft will remain safe without keeping the leader election quorum as a strict majority quorum. Raft also explicitly handles cluster membership changes as a special type of log entry. These two unique properties of Raft require leader election and cluster membership change entries to have strict majority quorums to keep safety intact. In the case that we do not take into account cluster membership changes, one can easily see the following as a split-brain scenario:

Let $N_t = 4$ (where t refers to the term of a leader). Applying relaxing of the majority quorums we get $|Q_{le}| = 3$ and $|Q_{lr}| = 2$. Say during this term t, a member is added to the cluster by a $|Q_{lr}|$. Now at $t + 1$ we have: $N_{t+1} = 5$, $|Qle| = 3$, $|Qlr| = 4$. Since we can now no longer guarantee that the new leader elected in $t + 1$ has the log entry of the new member being added (by quorum intersection), we can violate safety with the newly elected leader overwriting our commit marker.

The main benefits of using flexible quorums for Raft lies in the availability of even sized clusters. As mentioned earlier the typical Raft cluster size in practice is small; 5 or 7 nodes. We use odd sized clusters as they have better availability since each node in the system is less responsible for liveness. With flexible quorums we can show strict improvement of the availability of even sized clusters, which in practice are unavoidable due to membership changes, partitions, and crash-faults. We claim (refer to slides linked in appendix) that, with flexible quorums, an even sized cluster has the same asymptotic availability as an odd sized cluster with one less node. It is quite a trivial change to implement in practice, mainly because the original raft needs to be aware of cluster membership changes already. This means the only changes to the implementation are a relaxation of the log replication quorum when the flag of a cluster membership change is set. We will not describe further the implementation. See appendix for commit details.

3.2 Chained Logs

Using immutability is not a new concept, especially as it is becoming much more feasible with the cost of storage lowering. Making tradeoffs of memory footprint for performance is additionally favorable for large-scale cloud systems as they typically require much disk space in order to persist data regardless. As a result, we propose restructuring the raft commit log to be an immutable chain of blocks. We believe that this variant can increase observability and throughput under moderate leader failures because of the ability to send blocks of commands at the same time across the network.

The difference between original Raft and Chained Raft is that we will use a chain of "blocks" instead of a mutable log of events. Leaders add blocks to the head of their chain and replicate it across nodes. The result is each block storing a chain that cannot be changed, however forks can still occur. Resolving these forks can affect safety and is further discussed in the Implementation section. A major benefit of chained raft is that we do not need to track indexes in the log, just the last appended term and the head of the chain. Blocks currently store one entry but, in practice, will store chunks of entries. With moderate leader failures, original Raft will potentially change entries in its log to correct itself and ensure state machine safety. Under Chained Raft, blocks are appended to a chain much less frequently, since we put many commands into a block. Furthermore, the logs do not need to be corrected under leader failure and forks in the chain can be resolved faster, as described below. Before we could implement this immutable chain of blocks, we had to first ensure that the correctness and safety of the Raft algorithm would not be compromised. Recall the five properties that the Raft paper uses to prove safety: Election Safety, Leader Append-Only, Log Matching, Leader Completeness, State Machine Safety. We must define analogous properties for Chained Raft. Election Safety is the same. The Leader Append-Only property is that only a leader can add blocks to its head. The Log Matching property states that if two chains contain the same block then the path from the genesis block to the block on both chains will be the same. The Leader Completeness property states that if a block is committed in a given term then the heads of the leaders for all greater terms will either be the block or will extend the block. We will present a short proof of the Leader Completeness property as it is non-trivial and essential in ensuring safety in Chained Raft. Consider a block b that was first committed by the leader in term t. This means that at least a majority of servers had its last appended term as t and the heads of their respective chain are either b or extend b. We will do a proof by induction over the terms after t to show the Leader Completeness property holds.

Proof. Base case: Consider term $t+1$. A leader in term $t+1$ must have received votes from a majority and at least one of those servers has a last appended term of t and a head which is b or extends b. The leader cannot have a last appended term greater than t, so that means the leader must have t as its last appended term. So, its head is either b or extends block b. So, if there is a leader in term $t+1$, then b is present in the leader's chain.

Inductive case: Consider the term $t+k$. We assume that the property holds up to term $t+k-1$. A leader in term $t+k$ must have gotten a majority of votes to be elected. This means that at least one server's last appended term was, at some point, t and its head is either b or extends b. This server also must have voted for the leader. This means that any leaders since will have added blocks that extend b. Therefore, a leader in term $t+k$ has a head that is either block b or extends block b. □

4 Method

4.1 benchmarking

Our implementation began from a fork of an open-source Raft repository called eRaft. To implement flexible and chained raft, we wrote analogous functions that used chained RPC's and flexible quorums to replace the original core functionality in eRaft. Then, to test our system, we extended a client/server workbench that eRaft contributors provided. The measured workload is a distributed key-value store that puts 500,000 values across 10 client threads spread evenly across servers. The clients then output throughput and latency values for the valid puts. The clients perform validation by keeping track of requests and sending get requests. We ran our system in a pseudo-distributed fashion, with each node running as a docker container. We were unfortunately not able to mimic our benchmarks on a physical VM Cluster. We ensured that each server ran on seperate threads during benchmarking. To simulate realistic networked messages, we added 20ms delay to individual client requests.

4.2 Correctness Testing

The eRaft repo also had correctness tests that they ported from the original Raft paper. We converted those correctness tests to use blocks in order to validate the correctness of our chained implementation. Then, to test the fault tolerance of our raft implementation, we used slooo to programatically inject faults into our docker containers. These faults were as follows: CPU slow, lossy networks, slow networks, disk slow, and memory contention. We forewent crash faults, as time did not persist and we have no reason to believe the behavior would be notable.

5 Implementation

As noted earlier since the implementation of flexible quorum is trivial and not dependent on Chained Raft, it will not be covered here. The base of our implementation is forked from eraft, an open source generic raft library written in C++ linked in the appendix. We had to make changes to the structure of the specification for the Chained raft implementation to remain feasible. Following from the Chained Raft specification detailed prior, we have included additional information about the divergence of the specification from the implementation we utimately ended up using.

5.1 Block Markers

One of the main building blocks of Chained Raft is the idea of a block marker. The term marker is supposed to be analogous to a pointer, but in the case of a non-shared memory system (like a distributed system) this is quite an over-simplification. Thus, the main cost in changing from a mutable structured log entry to a immutable chain of blocks is the added overhead of finding equivalence for a atomic unit (block or entry). Indexes are no longer used, so we need to verify location as well as data to check for equivalence. To satisfy the safety property of consensus, we need to ensure there is no chance that a mis-equivalence will occur. This would cost us to have to walk the whole chain behind a block everytime, as each blocks identity is dependant on the blocks (chain) before it.

As a result, we propose the idea of giving up the absolute guarantee of safety, for a probabilistic guarantee. In practice we believe this tradeoff will allow the safety property to always hold true (although not provably), whilst still allowing the performance and observability gains that can result from the immutability changes to the log structure. We have seen other consensus algorithms (namely, PoW in Bitcoin[5]) utilize a similar tradeoff in practice, to massive practical success and safety. By attaching a 64 bit unsigned integer id to each block (generated by some pseudo random number generator), we can show that the probability of a mis-equivalence of two blocks is on the order of $\frac{1}{2}^{64}$. Under workloads with a large variety of unique commands, we can show much further reductions as we use id and command data for an equivalence check of two blocks. Furthermore, keeping track of all ids of blocks in active chains (any chain extending the commit block) allows the leader to ensure that no two blocks will be active with the same id. The only way this can break is if one leader generates the same id as the very next leader does. If the next leader does not see this id (less than $\frac{1}{2}$ nodes received the block) they may call a SendAppendRPC with an block with the same id that extends a different block. This would eventually lead to a safety violation. During our validation testing and benchmarking we were not able to observe such behavior, and think it would require network partitions or fail-stop restart faults. We should note that this tradeoff is implementation specific. In practice, we saw that nodes weren't noticeably slowed down when using true block chain validation. Thus, we abstract away whether or not the implementation uses true safety or probabilistic in the following sections.

5.2 Leader Commits

Without Chained blocks, the leader typically selects the $n/2$ highest match index (flexible quorum) of all its peers to be its updated commit index during a commit. With blocks, we change match index to "match offsets", where offsets refer to the closest block behind the head that the current peer has matched. The leader takes the $n/2$ lowest match offset as the potential new commit offset. The leader uses these offset to walk back from its head marker and update its commit block (if its a newer block). In other words, the leader then updates the commit block to the furthest block (longest-chain) that at least $n/2$ peers have matched. As a result of using offsets now, the leader needs to update the offsets every time its head marker moves.

5.3 Fork Garbage Collection

Upon an update of the commit marker, we can safely garbage collect any chains that no longer extend the commit block. Consequently, we call all chains that extend the commit block active as they still are needed for program execution. To garbage collect old forks, we keep track of all the active heads and their tails (the closest block on the commit chain) in a hash-map. Upon every update to the commit marker, we then garbage collect chains that are newly found "dead" (no longer extending the new commit block). This naturally allows the hash-map of alive chains to remain up to date and usable for it's intended searching purposes. We invoke garbage collection every update of the commit block, thus it is important to update the hash-map upon changes to the block chain.

5.4 Log Compaction

Our log compaction does not change from regular Raft. During compaction, we simply delete all blocks that are correspondingly being compacted from the block chain. Then we set the block which extends the persisted block to extend the genesis block (cut the tail). During update of commit marker, we convert block entries to regular log entries and keep our committed log state to be that of a regular raft log. This way we keep the immutability of the log while keeping the same interface for existing log compaction and entry persistance api's.

6 Results

As a result of our work, we believe to have found new motivation to continue pushing area in the research of Raft variants aimed at use in large-scale systems in practice. When we originally planned for this project, we were focused primarly about exploring the results of such work. Due to our cirucmstances however we were unable to gather the diverse and fine-grained data that we originally planned. The scope of the project was a lot larger than we intended, which caused most of our research to be based on the specification changes. However as a result of the broad scope of this project, we hope to motivate further work that is able to properly test and improve upon these ideas. We firmly believe that there is still much progress to be explored in the world of practical consensus.

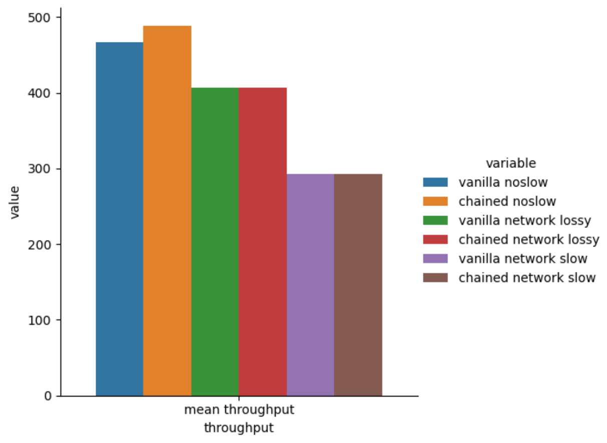

Figure 1. Mean Throughput

During our benchmarks, we were able to find slight throughput and latency gains for Flexible Chained Raft over Raft. We assume most of this is because of the flexible quorums, but we were also surprised to find our chained implementation not causing performance overhead. Chained blocks allow the leader to easily send all log entries to each node every RPC call. As soon as a follower finds the tail of the block in the message in its local block chain - it is able to append the whole chain. This decreases the messages the leader has to send, with the tradeoff of on average longer messages (and thus higher communication complexity overall). We also found that during implementation, it was quite easy to resolve bugs in the implementation due to the immutable structure of the block-chain. Overall, we are quite excited

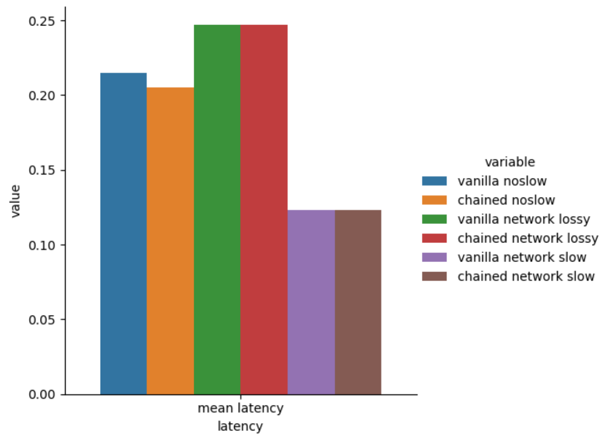

Figure 2. Mean Latency

in the results, specifically for the chained implementation, and believe that much further research can be done in this area. We hope to eventually extend and perhaps finalize this work in a later project where we can explore the behavior and benefits of the immutable log structure in finer details.

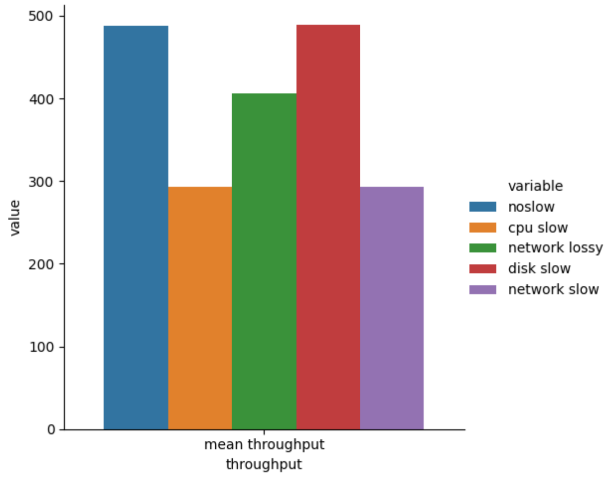

Figure 3. Mean Throughput for Chained Raft

During fault-injection benchmarks we saw that both implementations performed quite similarly under the more fine-grained faults that we tested. Specifically, under certain fail-slow faults (namely cpu slow and network slow) we saw massive throughput drops which equalized the implementations. This is because the clients that were communicating to the faulty nodes were no longer able to get valid responses in time, and thus achieved no throughput. Disk slow faults did not bottleneck the workload despite the heavy write workload, as for both implementations it's performance trended closely with no slow injections. This was interesting, but

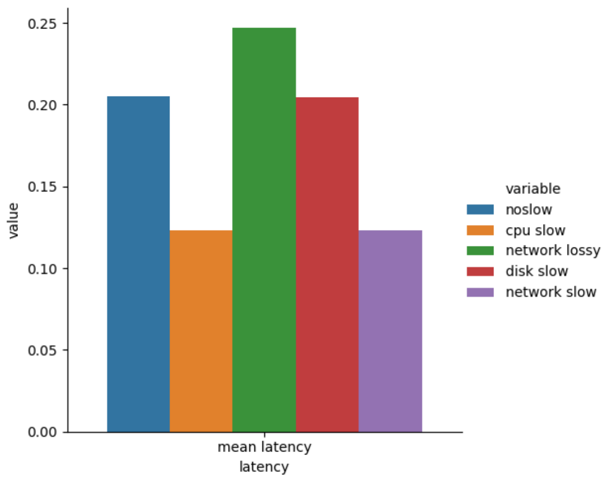

Figure 4. Mean Latency for Chained Raft

also gives us the idea that of all things to bottleneck in a distributed consensus node, it may be hardest to bottleneck disk write speed. There are much more frequent operations that are being executed, which are much more costly in terms of time. We imagine that with larger services with write heavy workloads built on top of the consensus cluster, such a fault would be much more damaging to performance.

Overall, we have seen that with flexible chained raft one can potentially see a performance gain under normal network behavior. We reasoned the potential benefits of using immutable block-chain based RPC's in Raft, both for easier implementation and observation.

7 Conclusion

This project is awesome.

8 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/Advai/eraft/tree/feat/add-blocks/

References

[1] Buterin, V., and Griffith, V. Casper the Friendly Finality Gadget. *CoRR abs/1710.09437* (2017).
[2] Howard, H. Simplifying raft with chaining, Jul 2021.
[3] Howard, H., Malkhi, D., and Spiegelman, A. Flexible Paxos: Quorum intersection revisited. *CoRR abs/1608.06696* (2016).
[4] Lamport, L. The part-time Parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.
[5] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.
[6] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[7] Wang, Z., Zhao, C., Mu, S., Chen, H., and Li, J. On the Parallels between Paxos and Raft, and How to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2019), Association for Computing Machinery, pp. 445–454.

[8] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. Hotstuff. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).