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 algorithms have been developed in research for simplicity and understandability. We find that certain 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 variations of Raft intended for better performance under specific, understudied fault models such as network partitions, and fail-stop restart 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 [**?**] and Paxos [**?**]. For this project, 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

Desirable qualities such as replication and consistency across large cloud-scale systems require the use of distributed consensus at some level. However, consensus is a very expensive procedure in terms of communication and round trip complexity. This results in most of the clusters running consensus algorithms in practice to be quite small, with larger clusters built on top of them that typically benefit from consistent reads that such consensus algorithms offer. This provides a degree of crash-fault tolerance, while minimizing the cost to latency and availability for users. For hyper-scalar cloud service providers, availability or client facing availability is the most commonly used metric for evaluating system effectiveness.

Raft was originally proposed in 2014 as an easier to understand and implement consensus algorithm when compared to Paxos. Despite those claims, there are many similarities between the two protocols. Specifically, research has shown that there exists a formal mapping between the two under moderate assumptions using refinement mapping [8]. The result of such work has given the ability to port specific optimizations from Paxos to Raft without sacrificing consensus requirements. This is crucial because there are many variations of Paxos, as it has been an active area of research for over twenty years now. In more recent years, new byzantine fault tolerant algorithms such as Tendermint, Casper FFG [7], and HotStuff [6] have utilized an immutable chain-based log structure which is a paradigm shift from the usual mutable log structures that Raft, Paxos, and PBFT use. Some less formal, raft-specific immutable log variants are also being explored in recent times [4]. As a result, we believe that there can be much research needed in Raft optimizations(?)

3 Solution

3.1 Flexible Quorums

One such variation of Paxos that has been proven to be applicable to Raft was Flexible Paxos [**?**]. 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

The related work of your project [**?**].

5 Implementation

As noted earlier the implementation of Flexible Raft is trivial and will not be covered here. The base of our implementation is forked from eraft, an open source generic raft library written in C++. We heavily changed the structure of the implementation for our chained raft implementation, which can be viewed in the repository linked below. Following from the Chained Raft specification detailed prior, we have included additional information about the divergence of the specification from the implementation we chose.

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. Mainly, we cannot check the equivalence of blocks by serializing the local virtual addresses in which they are stored, as there is no guarantee that each process is storing the blocks at the same address. The main cost in changing from a structured log entry to a chain of blocks is from finding equivalence of entries. If we take the naive approach and just use commands to check equivalence of blocks, we can run into a violation of safety by falsely claiming two blocks as equal. Yet if we are too cautious, we will have to walk (and thus also send) the whole chain of blocks preceding the block in order to guarantee equivalence. 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 safety 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[**?**]) utilize a similar tradeoff in practice.

The idea is as follows: if we attach a 64 bit 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 commands, we can show further reductions if we use comparison of block id, command type, and command in our equivalence check of two blocks. Lastly, we have a rough idea of how one could carefully implement such a system to ensure that the same 64 bit integer would have to be generated twice in a row (order of $\frac{1}{2}^{128}$).

Thus, we claim that in practice this probability should never be violated if we are careful. Mainly, we can store a local hash-map at each node keeping track of the ids in use. Once blocks have been committed (i.e the commit marker now extends the block), garbage-collected, or persisted, we can remove their id from the local map. When the leader then generates a new id for a new block they can ensure a unique id. This further reduces the chance at a collision. One may actually be able to show a formal proof that if we follow the above correctly, we can ensure that for a misclassification to occur, we have provided a rough idea in the appendix. The rest of our subsections build upon this idea of block equivalence. Thus, we abstract away whether or not the implementation uses true safety or probabilistic.

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 blocks behind the head that the current peer is. Now, as we take the $n/2$ lowest match offset, The leader uses this offset to walk back from its head marker and update its commit block. In other words, the leader then updates the commit block to the furthest block (longest-chain) that at least $n/2$ peers have found equivalence to.

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. To do this, we need to keep track of all the heads of each side chain of the block-chain using a vector of block pointers. The implementation of such is quite simple. We invoke garbage collection every update of the commit block, thus it is important to keep an accurate vector of chain heads.

5.4 Log Compaction

Lastly we wanted to note that we were not able to implement Chained Raft log compaction fully as of now. Instead, we will outline our plan for implementing compaction. Just like the original paper, Chained Raft will use snapshots for log compaction. Since blocks will be used to store a bulk of commands, we expect that snapshots will be taken less often and make log compaction. Storing snapshots will require walking a chain of block pointers, which will be less memory-effective as walking a contiguous block of memory. This project is still in progress for another week and a half, so we cannot speak for performance at this time.

6 Results

Alec will write here

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

References

[1] Howard, H., Malkhi, D., and Spiegelman, A. Flexible paxos: Quorum intersection revisited. *CoRR abs/1608.06696* (2016).

[2] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133:169.

[3] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.

[4] 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, p. 305:320.