FlexiRaft: Flexible Quorums with Raft

Ritwik Yadav
ritwikyadav@fb.com
Meta Platforms, Inc.
Menlo Park, California, USA

Anirban Rahut
arahut@fb.com
Meta Platforms, Inc.
Menlo Park, California, USA

Abstract

MySQL is the most popular transactional datastore deployed at Meta with a storage footprint in the order of petabytes. Over the years, several components have undergone significant changes to meet the demands posed by production workloads. One such effort was to redesign the replication protocol to use a modified version of Raft instead of traditional semisynchronous replication.

Even though Raft was a good fit for our requirements, the original algorithm did not offer much flexibility in choosing quorums which is important for latency sensitive applications. In this paper, we describe our changes to the original Raft algorithm required for supporting flexible data commit quorums. We discuss the impact of these changes on workload performance, fault tolerance and ease of integration into the existing production setup.

CCS Concepts: • **Computing methodologies** → **Distributed algorithms**; • **Information systems** → **Remote replication**.

Keywords: flexible quorums, consensus, raft, data replication

1 Introduction

MySQL is the transactional datastore of choice for relational workloads at Meta. The scale of MySQL deployment spans petabytes of data [19]. Over the years, lots of major improvements have been made to the MySQL stack [1] in order to support requirements stemming from serving production workloads. Some notable examples include development of middleware such as Binlog Server to efficiently provide in-region fault tolerance, a new storage engine [19] to reduce write amplification and several features to support multi-tenancy. One such effort was to redesign the replication protocol used by a multi-replica MySQL deployment to use Raft [22]. This paper describes the changes we made to Raft for supporting quorum flexibility and the lessons learned from the production deployment of these changes.

Every MySQL database in production has replicas in order to provide low latency reads across geographies. This redundancy also helps with fault tolerance. In the steady state, each database has a strong leader responsible for coordinating write operations to the database. Before the adoption of Raft, the consensus mechanism was split between the MySQL server and supporting automation tools. Modifying the code was error prone because the logic to support leader elections and data commit was spread across multiple bespoke automation tools. Crash recovery, leader election and disaster readiness exercises were all coordinated externally making it hard to reason about consistency and correctness of the protocol. During region outages (simulated or otherwise), the problem was even more exacerbated and significant manual effort was required to restore availability. In addition to that, clients used to be completely reliant on an external system to discover the primary replica serving write operations leading to scalability challenges in the past.

A redesign of the replication stack was undertaken to consolidate the logic into the MySQL server using a well defined consensus algorithm called Raft. Raft is an easy to understand consensus algorithm which is equivalent to Paxos [14] in fault-tolerance and performance [22]. It has strong leader semantics with clearly defined phases. There are lots of production grade open source implementations of the algorithm [2]. All of these properties made Raft a suitable candidate for implementing the next generation replication stack for our MySQL deployment. We had to modify the original Raft algorithm to eliminate performance bottlenecks and support configuration parameters which enabled developers to make the necessary tradeoffs for their applications. FlexiRaft is a direct result of these changes and some of its most important contributions are as follows.

- Data commit quorums were made configurable. The addition of flexible quorums enabled developers to make the necessary tradeoffs between latency, throughput and fault tolerance [7]. Leader election quorums get automatically computed from the specified data commit quorum to ensure correctness.
- Support for dynamic quorums was added wherein both the data commit and leader election quorums get reconfigured after every successful election. This option provides low latency commits with enhanced fault tolerance while restricting quorums to a small group of regionally local servers. Knowledge of the previous

data commit quorums is inferred from voting history. More details in subsection 4.2.
- Tail commit latencies became independent of the number of replicas in the cluster.
- Automation tools were significantly simplified since the consensus logic was completely incorporated into the MySQL server.

Section 2 provides some definitions to establish common terminology across different replication protocols. The preexisiting semisynchronous setup and potential solutions for its replacement are discussed in section 3. Section 4 of the paper describes the feature gaps in Raft and stresses on the need for flexibile quorums. It also lists the choices we provide to our end users when selecting configurable quorums followed by the amendments to the algorithm to support this flexibility in section 5. Section 6 discusses the fault tolerance guarantees of FlexiRaft with experimental validation of its performance presented in section 7 and lessons learned from its deployment in section 8. FlexiRaft is compared to other variants of consensus algorithms in section 9 along with a discussion on avenues for further improvement.

2 Common Concepts & Definitions

Some of the terms used in the paper are unique to the deployment of MySQL at Meta. This section provides definitions for these commonly used terms.

2.1 MySQL binlog server

The MySQL binlog server is a special server which only stores the recent binlogs (write ahead log for MySQL) rather than a full copy of the database. These special servers were developed at Meta to provide regional commits without incurring the overhead of extra replicas.

2.2 Replica set

A MySQL replica set is a collection of all the replicas (including the primary) and their corresponding binlog servers.

2.3 Group

Members of a replica set are grouped together into multiple disjoint sets based on physical proximity. Each disjoint set forms a group and physical proximity can be defined as belonging to the same region, same datacenter or sharing the same main switchboard (MSB) within a datacenter, etc. These groups are useful in defining quorums.

2.4 Data commit quorum

The data commit quorum is a minimal set of servers in the replica set (including both MySQL and binlog servers) that must acknowledge a transaction before it can be committed.

2.5 Leader election quorum

The leader election quorum is a minimal set of servers in the replica set (including both MySQL and binlog servers) that must accept a candidate server as a leader for it to *safely* assert leadership over the entire replica set.

2.6 Pessimistic quorum

A pessimistic leader election quorum consists of a majority of servers from *every* constituent group of a replica set. A majority in all constituent groups guarantees intersection with every valid leader election or data commit quorum. This is because every valid quorum is defined in terms of majorities in a subset of constituent groups.

3 Consensus Algorithms

A brief discussion on various consensus algorithms is presented in this section. The setup prior to Raft is also described in detail so that the rationale behind altering Raft to support quorum flexibility can be easily portrayed.

3.1 MySQL Semisynchronous Replication

Prior to Raft, our MySQL deployment utilized the semisynchronous replication protocol [6] in the data commit path to achieve durability. Figure 1 is a visual representation of a replica set which spans three regions. There is a MySQL database server in each of these regions. These servers are labeled $R0$, $R1$ and $R2$. Each MySQL server is accompanied by two binlog servers in the same region to provide local redundancy (fault tolerance) for the most recent write operations on the database. Binlog servers labeled BLS $R0.1$ and $R0.2$ acknowledge transactions from MySQL server $R0$, binlog servers BLS $R1.1$ and $R1.2$ acknowledge transactions from MySQL server $R1$ and so on. As pointed out in subsection 2.1, MySQL servers have the state machine as well as the write ahead log whereas the binlog servers only have the write ahead log. Only the primary replica accepts writes to the database (strong leader semantics). The semisynchronous unit consists of the primary[1] replica and its two binlog servers. For a transaction to get committed, one of the binlog servers out of BLS $R0.1$ and $R0.2$ would need to acknowledge it to $R0$. The rest of the replicas ($R1$ and $R2$) would asynchronously apply updates to the database. It is to be noted that binlog servers BLS $R1.1$ and $R1.2$ get updates from $R1$ and not from $R0$ directly. Similarly, binlog servers BLS $R2.1$ and $R2.2$ only get updates from $R2$ (not $R0$).

If the primary ($R0$) failed, a different replica would take its place and use the failed primary's binlog servers (BLS $R0.1$ & $R0.2$) to apply any committed operations that it couldn't replicate before the failure. The leader election process was externally orchestrated by monitoring processes running on database hosts. The separation of logic for committing data and electing a new leader was error prone and difficult

[1]The terms $primary$ and $leader$ are used interchangeably in this paper.

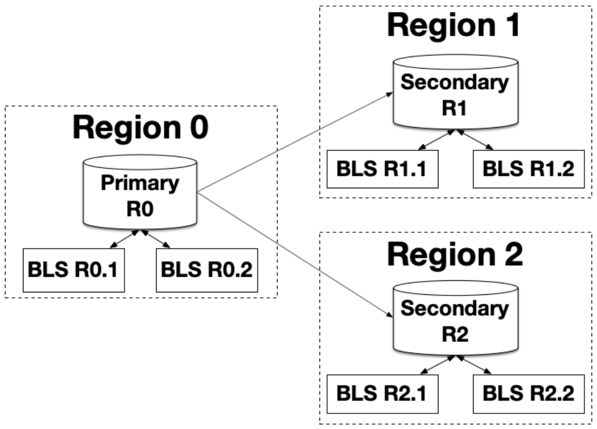

Figure 1. Sample replica set with members in three geographical regions

to maintain. This led to a redesign of the replication stack and the Raft consensus algorithm was employed to commit entries to MySQL's write ahead log [4]. The implementation of the algorithm was extracted [3] from the Apache Kudu [18] open source project.

3.2 MySQL Group Replication (Paxos variant)

Another significant and deliberate choice was to *not* use group replication offered by MySQL version 5.7 onwards. While there are significant advancements offered by the protocol (such as the multi-primary mode), using group replication in our deployment presented significant challenges. It is based on a variant of Paxos [14] which does not use a persistent log. Entries are written to the log only after they are considered committed via in-memory consensus rounds. The leader election algorithm [5] is local and deterministic. It also doesn't consider lag when choosing a new leader and brief network blips trigger a computationally expensive recovery algorithm. This option could have worked but not without excessive engineering effort to fix the drawbacks.

3.3 Raft

Raft is a consensus algorithm which is equivalent to (multi-) Paxos in fault-tolerance and performance [22]. It was designed with the primary goal of improving understandability and is described completely enough to meet the needs of a practical system. Compared to Paxos, it offers state space reduction and decomposes consensus into well defined phases for leader election, log replication and safety.

The Raft algorithm achieves consensus via an elected leader and is *not* a Byzantine fault tolerant algorithm. It imposes the restriction that only the servers with the most up-to-date data can become leaders and includes a new mechanism for changing cluster membership by utilizing overlapping majorities to guarantee safety. A formal proof of correctness for its consensus mechanism has been provided using the TLA+ framework [21].

Besides its advantageous simplicity, Raft has several commonalities with the previous semisynchronous deployment because of its single strong leader semantics and fault tolerance guarantees. In addition to that, availability of several well-tested open source implementations of the algorithm made it a suitable candidate for adoption in our revised replication framework despite some feature gaps explained in the following sections.

4 Flexible Quorums

Two shortcomings of replacing the semisynchronous setup with the original Raft algorithm are immediately obvious. First, it would increase the cross region network utilization as the leader (primary replica) would directly send updates to all other members of the replica set. Please note that binlog servers of every region received updates from their local MySQL replica in the semisynchronous protocol (subsection 3.1). In case of Raft, the primary replica $R0$ would be directly sending updates to binlog servers BLS $R1.1$, $R1.2$, $R2.1$ and $R2.2$, increasing the traffic between region pairs (region 0, region 1) and (region 0, region 2) shown in Figure 1. Second, a majority of the replica set participants would need to agree before committing any write operations to the database. This would significantly increase data commit latency and reduce overall throughput because the agreement would cross regional boundaries. In the example shown in Figure 1, there are 9 members in the replica set and at least 5 votes are required by the primary server $R0$ to commit any transaction. Since region 0 only has 3 members, 2 votes (out of the 5 required) would be needed from servers outside of region 0. These regressions would present challenges for most workloads we run in production. In this paper, we only discuss mitigation for increased data commit latencies by adding flexible quorums to Raft. The idea is similar to Flexible Paxos [12]. The mitigation strategy used to reduce cross-region network utilization is beyond the scope of this paper.

We offer flexibility to end users when selecting data commit quorums since it determines the latency, throughput and fault tolerance observed by their application. Valid leader election quorums are automatically computed from the chosen data commit quorums. For state machine safety [22], every data commit quorum needs to intersect with every leader election quorum. If this constraint is violated, two disjoint sets of servers would exist where one set would be able to commit transactions without the knowledge of any member in the other set which is capable of independently electing a new leader (data loss). Similarly, since our setup only allows for at most one leader at any given point in time, every pair of valid leader election quorums should intersect as well. If this were not the case, there would exist two disjoint sets of servers wherein each set is capable of independently electing a new leader without the knowledge of any member in the other set (split brain).

The most common design choice for applications is to choose a regionally local quorum and compromise on availability if the entire region were to go down. This was the only available option with the semisynchronous protocol. In the example replica set shown in Figure 1, write availability would be lost if region 0 becomes unavailable. In fact, only two servers out of MySQL server $R0$, binlog servers BLS $R0.1$ and $R0.2$ need to become unresponsive for the replica set to lose write availability. With FlexiRaft, the exact same configuration is offered by the dynamic quorum mode. FlexiRaft supports two different modes when choosing data commit quorums; static mode and dynamic mode. Data commit quorums that span over multiple geographical regions are configured with the static mode.

4.1 Static Quorums

As the name suggests, static quorums can be enumerated deterministically and are not a function of which group the leader belongs to. Enumeration involves listing all possible subsets of servers that need to agree for committing write operations. The helper function is_static used in algorithm 1 checks the configuration of a replica set to return a boolean value denoting whether the data commit quorum is static.

Two examples of specifying static quorums are as follows.

4.1.1 Disjunction.

This option is useful for replica sets that span over multiple geographies. One such example is where the leader could reside on either side of the Atlantic Ocean depending on the time of the day. Assuming an application needs to commit data only after it has been agreed upon either by two datacenters in the United States or two datacenters in Europe, the data commit quorum could be configured as follows.

```
Majority in 2 out of 5 groups: {G1, G2, ..., G5}
                              OR
Majority in 2 out of 3 groups: {G6, G7, G8}
```

G1, G2, G3, G4 and G5 are disjoint groups of servers in the United States and G6, G7 and G8 are disjoint groups of servers in Europe. These groups are based on datacenter membership, i.e., all servers in a group belong to the same datacenter. The corresponding leader election quorum would be the following:

```
Majority in 4 out of 5 groups: {G1, G2, ..., G5}
                              AND
Majority in 2 out of 3 groups: {G6, G7, G8}
```

4.1.2 Conjunction.

The following example specification can provide stronger consistency guarantees across geographies where we require members in two datacenters on each coast of the United States to agree before committing data. It would be specified like the following:

```
Majority in 2 out of 5 groups: {G1, G2, ..., G5}
                              AND
Majority in 2 out of 3 groups: {G6, G7, G8}
```

G1, G2, G3, G4 and G5 are disjoint groups of servers on the east coast of the United States and G6, G7 and G8 are disjoint groups of servers on the west coast. Similar to the disjunction example, these groups are based on datacenter membership. The corresponding leader election quorum which gets automatically computed is as follows.

```
Majority in 4 out of 5 groups: {G1, G2, ..., G5}
                              AND
Majority in 2 out of 3 groups: {G6, G7, G8}
```

It is trivial to see that in each of these cases any leader election quorum would intersect with any data commit quorum and any pair of potential leader election quorums would also intersect.

4.2 Dynamic Quorums

In this mode, the data commit quorum is limited to only one group. Groups are usually defined by a geographical region (eg. Prineville, OR). Both the data commit and leader election quorums are expressed as majority in the group containing the current leader. Therefore, the quorums are dependent on which member of the replica set is currently serving as the leader. This is a popular choice in our setup today.

If we were to implement single region data commit quorum using the static mode defined above, leader election would require a majority to be available in *all* groups (pessimistic quorum) as any of the constituent groups of the replica set could have the previous leader. This can be a problem if the leader dies and one of the groups (not necessarily that of the leader) is unavailable due to any reason - actual network connectivity issue, disaster readiness exercise etc. We modified the Raft algorithm to make it possible for candidates to learn about the previous leader without necessarily having to consult majorities in all groups.

5 Algorithm

Changing the original Raft algorithm to support the static quorum modes was relatively less intrusive. Instead of waiting on a simple majority of voters, the algorithm would now evaluate a configurable static quorum instead. The helper function $is\_quorum\_satisfied$ mentioned in algorithm 1 returns a couple of boolean values after analyzing all the votes received by the candidate. The first boolean denotes if the quorum has already been satisfied from the votes received thus far and the second one denotes if the satisfaction of the quorum is still possible. However, for supporting the dynamic quorum mode, the leader election algorithm had to undergo significant changes described in the subsections which follow.

Algorithm 1: FlexiRaft Leader Election

```
Input: $responses : set < RequestVoteResponse >, quorum\_specification, last\_known\_leader$
Output: ElectionResult<$quorum\_satisfied, quorum\_satisfaction\_possible$>
1  if $is\_static(quorum\_specification)$ then
2      <$quorum\_satisfied, quorum\_satisfaction\_possible$> ← $is\_quorum\_satisfied(responses, quorum\_specification)$
3      return ElectionResult<$quorum\_satisfied, quorum\_satisfaction\_possible$>
4  end
   /* Otherwise, executing in dynamic mode                                    */
5  <$quorum\_satisfied, quorum\_satisfaction\_possible$> ← $is\_quorum\_satisfied(responses, pessimistic\_quorum)$
6  if $quorum\_satisfied$ or $least\_known\_leader = NULL$ then
       /* last_known_leader could be NULL before the first successful election  */
7      return ElectionResult<$quorum\_satisfied, quorum\_satisfaction\_possible$>
8  end
9  if $current\_term = last\_known\_leader.term + 1$ then
10     $quorum \leftarrow majority(last\_known\_leader.group)$
11     <$quorum\_satisfied, quorum\_satisfaction\_possible$> ← $is\_quorum\_satisfied(responses, quorum)$
12     return ElectionResult<$quorum\_satisfied, quorum\_satisfaction\_possible$>
13 end
14 $term\_it \leftarrow last\_known\_leader.term$
15 $next\_leader\_groups \leftarrow \{last\_known\_leader.group\}$
16 $explored\_leader\_groups \leftarrow \{last\_known\_leader.group\}$
17 $terminal\_groups \leftarrow \phi$
   /* Iterate until the pessimistic quorum is absolutely required.            */
18 while $explored\_leader\_groups \subset \mathcal{G}$ do
19     $status, next\_term, potential\_next\_leaders \leftarrow GetPotentialNextLeaders(term\_it, next\_leader\_groups)$
20     switch $status$ do
21         case $ALL\_INTERMEDIATE\_TERMS\_DEFUNCT$ do
22             $quorum\_results \leftarrow \{is\_quorum\_satisfied(responses, majority(group)) : group \in terminal\_groups\}$
23             return < $\bigwedge_{qr \in quorum\_results} qr.quorum\_satisfied, \bigvee_{qr \in quorum\_results} qr.quorum\_satisfaction\_possible$ >
24         end
25         case $WAITING\_FOR\_MORE\_VOTES$ do
26             return $ELECTION\_UNDECIDED < false, true >$
27         end
28         case $POTENTIAL\_NEXT\_LEADERS\_DETECTED$ do
29             $term\_it \leftarrow next\_term$
30             $next\_leader\_groups \leftarrow \bigcup_{next\_leader \in potential\_next\_leaders} \{next\_leader.group\}$
31             $explored\_leader\_groups \leftarrow explored\_leader\_groups \cup next\_leader\_groups$
32         end
33     end
34 end
35 return $ELECTION\_UNDECIDED < false, true >$
```

5.1 Extra State

There are two extra pieces of information that each server would need to store for supporting dynamic quorums.

- Last known leader and its term
- Previous voting history

5.2 Leader Election

Algorithm 1 describes the modifications to the original Raft leader election algorithm [22] to support the quorum modes described in section 4.

At a high level, the algorithm first checks if the the configured data commit (and hence, the leader election) quorum is static and returns if the quorum has been satisfied based on the votes received thus far. In dynamic mode, the algorithm checks for the satisfaction of the *pessimistic_quorum*. The *pessimistic_quorum* for a replica set is defined as a majority within *all* groups of servers individually. By definition, satisfaction of the *pessimistic_quorum* guarantees the satisfaction of the leader election quorum (since it is defined as majority in a subset of the groups). If the candidate does not have any knowledge of the last known leader, the only way to win an election is to satisfy the *pessimistic_quorum*. If the term in which the candidate is conducting an election immediately succeeds that of the last known leader, getting a majority from the group of the last known leader is sufficient to win the election. Otherwise, the candidate now needs to learn about the leader (if it exists) with the highest term and solicit majority votes from that leader's group.

The leader with the highest term must have won an election after the last leader known to the candidate. The algorithm tries to figure out the identity of this leader with the highest term from the voting history of the replicas. The voting history of the replicas is included in the *RequestVoteResponse* RPC sent to the candidate. The *GetPotentialNextLeaders* subroutine (algorithm 2) helps in determining the leader with the highest term.

GetPotentialNextLeaders returns a 3-tuple as it's output. The first element is a status indicator which could take one of three values. If a majority of the replicas in a group which could potentially have the latest leader have not responeded, a decision cannot be made and status code *WAITING_FOR_MORE_VOTES* is returned. All the terms between the term of the last known leader and the candidate's current term are iterated upon (in an increasing order) to determine if some replica could have potentially won an election in those intermediary terms. Status code *ALL_INTERMEDIATE_TERMS_DEFUNCT* is returned when the set of all servers which could have potentially won an election without the candidate's knowledge has been determined. At this stage, the candidate can win an election if it has a majority vote from each of these potential leaders' groups (termed as *terminal_groups*). If potential election winners in the intermediary terms are found, then status code *POTENTIAL_NEXT_LEADERS_DETECTED* is returned and the iteration continues until majority is achieved in all terminal groups, the quorum converges to the *pessimistic_quorum* (terminal groups include all groups, denoted by $\mathcal{G}$) or the election times out.

The *potential_election_winners* is a subroutine that takes a historical term t as input and looks at the voting histories of the replicas to determine the set of all potential leaders in term t. The implementation of this subroutine and a few others mentioned in the paper such as *majority*, *majority_count* and *vote_count* is trivial and not included in the paper. Full implementation and all of the source code is publicly available [3] . The FlexiRaft algorithm is validated

Algorithm 2: GetPotentialNextLeaders

```
   Input: $term\_it, possible\_leader\_groups$
   Output: $status, next\_term, potential\_next\_leaders$
   Data: $responses : set < RequestVoteResponse >$
1  if $\exists\, group \in possible\_leader\_groups :$
     $majority\_count(group) > vote\_count(group)$ then
       /* Majority of the votes haven't arrived
          in a group which could potentially
          have the leader.                        */
2      return
        $< WAITING\_FOR\_MORE\_VOTES, -1, \phi >$
3  end
4  do
       /* Compute the set of all historical
          votes across all voters.                */
5      $historical\_votes \leftarrow \{r.previous\_vote\_history :$
        $r \in responses\}$
6      $min\_term \leftarrow min(\{vote.term$ if
        $vote.term > term\_it : vote \in historical\_votes\})$
7      $possible\_leaders \leftarrow$
        $potential\_election\_winners(min\_term,$
        $possible\_leader\_groups, responses)$
       /* Updating terminal_groups is a side
          effect of this subroutine.              */
8      $terminal\_groups \leftarrow$
        $terminal\_groups \cup possible\_leader\_groups$
9      if $possible\_leaders \neq \phi$ and
        $min\_term < current\_term$ then
10         return
            $<POTENTIAL\_NEXT\_LEADERS\_DETECTED,$
            $min\_term, possible\_leaders>$
11     end
12 while $min\_term < current\_term$
13 return <
     $ALL\_INTERMEDIATE\_TERMS\_DEFUNCT, -1, \phi >$
```

by a TLA+ specification[2] which is also included in our open source project.

6 Fault tolerance

The general idea is to shrink the data commit quorum at the expense of expanding the leader election quorum. It proves to be a good compromise since committing data is a more frequent operation [12]. As described in subsection 4.1, it is easy to choose quorums in static mode that are resilient to the failure of an entire group. This group could be an entire datacenter. Further discussing the example of a static quorum presented in subsubsection 4.1.1, it is evident that failure of any group will not result in availability loss even

[2]https://github.com/facebook/kuduraft/blob/1.8.raft/tla/flexiraft.tla

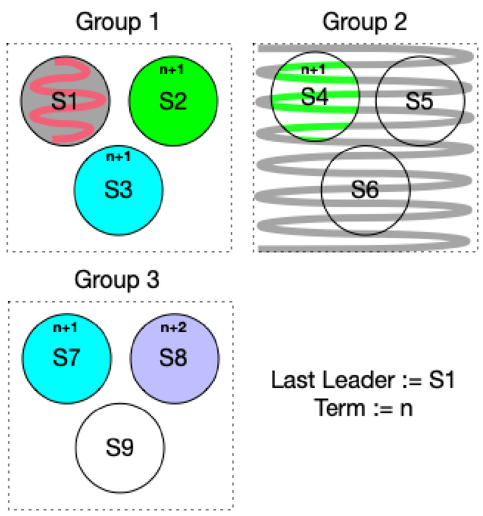

Figure 2. Leader election failure in dynamic mode of FlexiRaft

if that group contains the leader. Without loss of generality, we can assume that group $G6$ contains the leader and it gets partitioned off from the rest of the members in the replica set. Group $G6$ cannot commit any transactions anymore without a majority acknowledgement from either group $G7$ or $G8$ as per the configured data commit quorum. Once the servers outside of $G6$ timeout after failing to receive a hearbeat from the leader in $G6$, one of the servers would be able to win leadership without getting votes from any member of $G6$. This is because for leader election, the requirement is to get a majority vote from 2 out of the 3 groups; $G6$, $G7$ and $G8$. Once a new leader is elected, it can start commiting transactions without getting a single vote from any member in group $G6$ as per the configured data commit quorum. Therefore, the availability of the replica set remains unaffected despite the failure of an entire group. The same logic applies for the example in subsubsection 4.1.2.

A FlexiRaft replica set configured to use dynamic quorums may not be resilient to failure of a majority of servers in one group. One obvious example is when a majority of the servers in the group containing the leader fail, disrupting both the data commit and leader election quorums simultaneously. However, there exist other failure scenarios with the dynamic quorum configuration where a coordinated failure of the leader and a majority of the servers in a group not containing the leader can also lead to availability loss. This is demonstrated in Figure 2 where a replica set with three groups of servers is configured to use dynamic mode. Both the data commit and leader election quorums would be a majority of servers in group 1 since the leader is server S1 (in term n). Consider the following plausible sequence of events which lead to availability loss.

- Servers $S4$ and $S7$ both initiate an election in term $n+1$ because of a temporary network disruption which prevented them from receiving a hearbeat from server $S1$ before the timeout. Both servers $S4$ and $S7$ are fully caught up with server $S1$.
- Server $S4$ gets server $S2$'s vote and server $S7$ gets server $S3$'s vote in term $n+1$. Servers $S2$ and $S3$ give their votes because the election is being conducted in a higher term and both candidates are sufficiently caught up.
- Server $S1$ fails but a majority of the servers in group 1 are still functioning properly.
- Group 2 gets cut-off from the rest of the servers in the cluster due to network partitioning.
- Neither server $S4$ nor server $S7$ wins the election in term $n + 1$.

Now, if server $S8$ times out and starts an election in term $n + 2$, it cannot safely win the election until $S1$ or group 2 come online. This is because $S1$ could have given its vote to $S4$ making it the leader in term $n + 1$ and there is no way for $S8$ to verify that. Therefore, availability of a replica set configured to use dynamic mode of FlexiRaft might get affected because of coordinated failures of the leader and a group *not* containing the leader. Situations such as this are prevented by the pre-vote algorithm [21].

7 Experimental Results

Two main aspects of FlexiRaft's performance are demonstrated by our experiments. Firstly, the throughput and client-observed latencies are comparable for FlexiRaft and semisynchronous algorithms. In all of our experiments, FlexiRaft was deployed in the dynamic mode with data commit quorum restricted to a single group. Grouping of servers was done based on the datacenters they belonged to. The same replica set with the same machines were used for all three consensus algorithms. Each test machine was configured with a dual socket Intel Xeon Gold 6138 CPU and 256 GB DDR4 RAM. Each socket offered 20 physical cores.

Figure 3 shows the throughput as we increase the number of clients writing to the database. Throughput is computed by measuring the amount of time taken to commit 2 million transactions generated by our microbenchmark. The amount of data written by the transactions in this microbenchmark range anywhere from 32 bytes to 256 bytes and roughly half of them have primary key based updates alongside newly inserted rows.

Latency measurements are specified in Table 1 and were computed across a million uniform commit samples for each algorithm. It can be seen that the average numbers for FlexiRaft[3] and semisynchronous algorithms are similar.

Secondly, unlike Raft, the performance of FlexiRaft[3] does not deteriorate with an increasing number of servers in the replica set. Figure 4 shows a graph between average latency

[3]dynamic mode

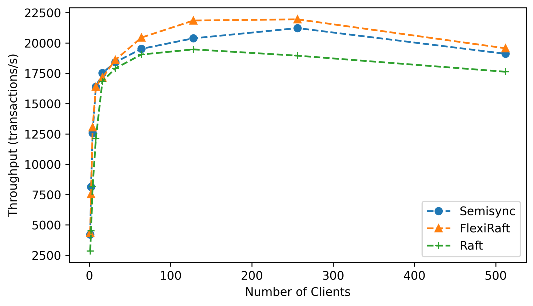

Figure 3. Comparison of throughput observed. Throughput is expressed as transactions per second.

Table 1. Latency comparison between different algorithms

Algorithm	Minimum Latency	Maximum Latency	Average Latency
Semisync	8ms	102ms	12ms
FlexiRaft	9ms	98ms	12ms
Raft	19ms	156ms	29ms

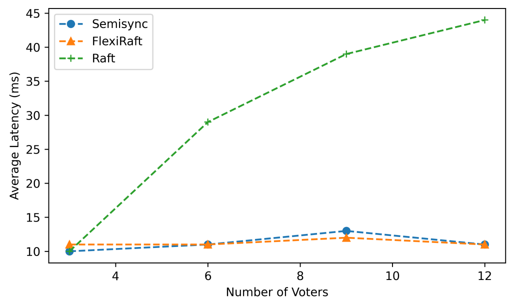

Figure 4. Effect on latency with increasing replica set size

observed by clients for each of these algorithms. The number of voters was increased from 3 to 12 and the effect on average commit latency was measured. As can be seen from the graph, Raft's latency goes up with an increase in the replica set size because of the corresponding growth in data commit quorum size. However, for semisynchronous and FlexiRaft[3], it remains stable since the quorum is confined to a single group of servers belonging to the same datacenter.

8 Lessons Learned

Numerous variants [8, 9, 11–13, 15–17, 20] of Paxos have been developed over time to customize different aspects of performance and application requirements. While the idea behind flexible quorums has been applied to Paxos before [12] and the underlying compromises are straightforward to understand, the details around implementation play an equally significant role in determining performance and enhancing understandability. Flexible quorums allow a smaller data commit quorum which can be arbitrarily chosen but there need to be guardrails / workflows for end users that simplify the selection of their data commit quorums based on their application's need. This was the reason why we chose to offer dynamic mode to cater to the users that got accustomed to the semisynchronous protocol over time and built the static mode with varying degrees of fault tolerance for users who wanted better fault tolerance and / or consistency guarantees. All quorums are defined in terms of majorities within groups of servers in the replica set to keep the logic simple. Initially, groups were pre-determined based on the datacenter each server belonged to. Later, group definitions were made generic enough to support varying levels of physical proximity as described in subsection 2.3.

A number of optimizations were landed in kuduraft [3] to remove unnecessary contention and achieve performance at par with the semisynchronous replication protocol. Most of these optimizations prevented severe degradations to the observed tail latencies and are generic enough to be applied to any consensus algorithm. Some examples include making local vote counting logic asynchronous so that it wouldn't block threads writing to the log, quorum aware optimizations to advance watermarks, etc. Quorum awareness can be used to optimize multiple bottlenecks in the data commit path. For example, the procedure to advance commit watermark can be skipped upon receiving a vote from a follower which is not a member of the data commit quorum. This removes unnecessary contention by reducing the time for which locks are held. Another important issue that was hurting tail commit latencies was the repeated initiation of pre-elections from old members of the replica set. There are multiple ways to solve this problem originating from former members disrupting the operations of an otherwise healthy replica set. Specialized response codes can be sent back to former members or necessary cleanups can be done externally by automation tools upon eviction of any member from the replica set.

There are some important features described in the formal specification of Raft [21] which are not mandatory from a correctness standpoint but make life easier when managing any deployment of a Raft variant at scale. Pre-voting (or pre-election) is one such feature which became more important with the addition of flexible quorums. In the pre-vote algorithm, a candidate only increments its term if it first learns from the memebers of a leader election quorum that they would be willing to grant the candidate their votes. This helps reduce the likelihood of disruptions, especially in the dynamic quorum mode where coordinated failures of the leader and a group not containing the leader can cause availability loss for the replica set as described in section 6. Another crucial improvement facilitating multiple cluster membership changes simultaneously is the joint consensus algorithm. It defines a transitional configuration where agreement requires quorum satisfaction from both the new configuration and the old configuration. This approach extends the single server membership change algorithm which was deemed sufficient for production deployments since any arbitrary change can be implemented as a series of single server membership changes [21]. However, in practice, joint consensus simplifies operational tooling to a significant extent and makes routine tasks (such as maintenance drains, disaster recovery exercises, etc.) efficient and easier to manage at scale.

9 Related and Future Work

Over the last two decades, numerous variants [8–13, 15–17, 20, 23, 25] of consensus algorithms have been developed to solve for scalability bottlenecks. Three main avenues of improvements are usually targeted by these variants. The first improvement comes from the realization that the data commit quorum is exercized more often than the leader election quorum. Therefore, reducing the size of the data commit quorum at the expense of expanding the leader election quorum is a viable tradeoff [12] which leads to improved throughput and lower latencies. To the best of our knowledge, FlexiRaft is one of the first large scale deployments of flexible quorums to Raft.

A large number of variants target the skew in the amount of work that is performed by the primary as compared to the read-only replicas. Varying solutions have been studied over time which make different compromises. The thrifty protocols [17, 20] only contact a minimal quorum of nodes rather than sending updates to all replicas (followers) but risk the performance getting affected by a single sluggish replica. Variants such as WPaxos [8] distribute the work across multiple leaders to achieve scalability and employ an adaptive object stealing protocol to match the workload access locality. Compartmentalized protocols [25] are similar to Raft in the sense that they breakdown the consensus problem into multiple distinct roles and allocate these roles to different servers. Leadership is delegated to *proxy-leaders* for each command. FlexiRaft doesn't attempt to solve the leader bottleneck but its dynamic quorum mode is similar to Mencius [9] since the leader can rotate between different members of the replica set and quorums get adjusted automatically upon every successful leader election.

Another significant avenue for improvement is when strict serializability and consistency can be relaxed in favor of low latency and high throughput. The ParallelRaft algorithm implemented by PolarFS [10] makes this tradeoff. It relaxes Raft's strict serialization by allowing out-of-order acknowledgement, commit and application of logs. Since MySQL doesn't care about the I/O sequence of underlying storage, exploring this strategy for FlexiRaft to further provide a throughput boost for applications could be a fruitful endeavor in the future.

All three classes of improvements discussed above can be applied together depending on the needs of the application. In fact there have been studies [24] to facilitate porting improvements from different variants of Paxos to Raft.

Acknowledgments

We sincerely thank all our CIDR 2023 reviewers for their insightful feedback and suggestions. We also appreciate the efforts of Kai Zeng for shepherding our paper and providing valuable guidance towards its improvement. The project would not have been possible without the hard work of the Database Engineering and MySQL Infrastructure groups at Meta. We would especially like to extend our gratitude to Shrikanth Shankar, Bartłomiej Pelc, Yichen Shen, Vinaykumar Bhat and Michael Percy for their numerous contributions and design discussions.

References

[1] Github. *Facebook's branch of the Oracle MySQL v5.6 database.* https://github.com/facebook/mysql-5.6
[2] Github. *The Raft Consensus Algorithm.* https://raft.github.io/
[3] Github. *A Raft Library in C++ based on the Raft implementation in Apache Kudu.* https://github.com/facebook/kuduraft
[4] MySQL Internals Manual. *Binary Log Overview.* https://dev.mysql.com/doc/internals/en/binary-log-overview.html
[5] MySQL Reference Manual 8.0. *Group Replication: Single-Primary Mode.* https://dev.mysql.com/doc/refman/8.0/en/group-replication-single-primary-mode.html
[6] MySQL Reference Manual 8.0. *Semisynchronous Replication.* https://dev.mysql.com/doc/refman/8.0/en/replication-semisync.html
[7] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.
[8] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.
[9] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08).*
[10] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.
[11] Eli Gafni and Leslie Lamport. 2003. Disk paxos. *Distributed Computing* 16, 1 (2003), 1–20.
[12] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696* (2016).
[13] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.
[14] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport.* 277–317.
[15] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. Stoppable paxos. *TechReport, Microsoft Research* (2008).
[16] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing.* 312–313.

[17] Leslie Lamport and Mike Massa. 2004. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 307–314.

[18] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. 2015. Kudu: Storage for fast analytics on fast data. *Cloudera, inc* 28 (2015).

[19] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[20] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2012. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*.

[21] Diego Ongaro. 2014. *Consensus: Bridging theory and practice*. Stanford University.

[22] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[23] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. 2017. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 426–438.

[24] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and how to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 445–454.

[25] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2203–2215.