MyRaft: High Availability in MySQL using Raft

Anirban Rahut, Vinaykumar Bhat, Abhinav Sharma, Yichen Shen, Bartlomiej Pelc, Chi Li, Ahsanul Haque, Yash Botadra, Xi Wang, Michael Percy, Ritwik Yadav, Yoshinori Matsunobu, Alan Liang, Igor Pozgaj, Tobias Asplund, Anatoly Karp, Luqun Lou, Pushap Goyal*
Meta Platforms
Menlo Park, California, United States
myraft-paper@meta.com

ABSTRACT

Meta uses MySQL to manage tens of petabytes of data for various internal services including our largest database that serves the social graph. In the past, we used a mix of MySQL's semi-synchronous and asynchronous replication protocols. We relied on external processes for control plane operations, like failover and cluster membership changes, to provide high availability and fault tolerance. This system was complicated, prone to edge cases, and hard to prove theoretically. Therefore, we decided to transition our MySQL replication stack to a proven consensus protocol, namely Raft. In this paper we present *MyRaft*, a MySQL server integrated with Raft. The integration was achieved through a MySQL plugin, along with necessary modifications to MySQL replication. In order to meet demands for low latency and high availability in our production deployment, we made multiple enhancements to Raft, e.g. FlexiRaft and Proxying.

We have migrated a large portion of our deployment to MyRaft. This has helped reduce MySQL failover time from one minute to a few seconds, while still achieving low commit latency. In this paper we describe how the integration was achieved to make MySQL a natively fault-tolerant distributed database. We also share our experience and insights from rolling out MyRaft at scale.

1 INTRODUCTION

1.1 Motivation

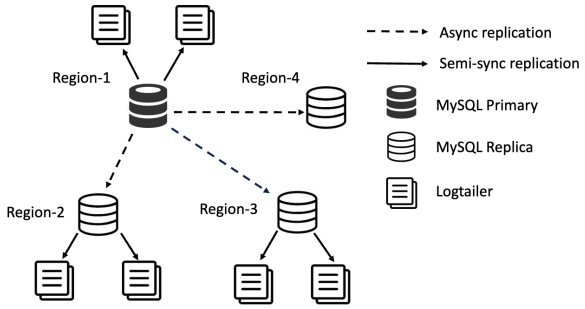
Historically, RDBMSs like MySQL were built to run on a single node. As the need for fault tolerance and high availability increased, single node databases evolved to multi node distributed systems. Starting in the early 2000s, several influential papers on distributed systems were published, including the Google File System [13], BigTable [9], Cassandra [17], and Spanner [12]. The foundation of many of these systems was a consensus algorithm like Paxos [18], which offers consistency even in the event of failures.

At Meta, MySQL is the most commonly used OLTP database, serving as the relational data store for services like the social graph. Our MySQL deployments are geo-replicated for fault tolerance, high-availability and scalability. A cluster of replicated MySQL servers and log-only entities (called *logtailers*) is called a *replicaset* (Figure 1a). Each replicaset contains a set of shards which map one-to-one to MySQL databases. Every replicaset has a single primary instance (at a given time) that handles all write transactions. These transactions are asynchronously replicated to replicas using MySQL's binary log [21]. Not all replicas are capable of becoming the primary. We attach logtailers to every primary-capable replica. Logtailers are special servers that do not have a storage engine; they can only receive, store, and send replication logs. MySQL's semi-synchronous [1][36] replication protocol was previously used between the primary instance and the logtailers. The logtailers send acknowledgement to the primary after they have persisted the transactions in the local storage. Typically, a primary has two logtailers attached to it, and it waits for acknowledgement from at least one logtailer before committing a transaction. This makes the transaction durable. In case the primary is unavailable, the logtailers can help elect a new primary.

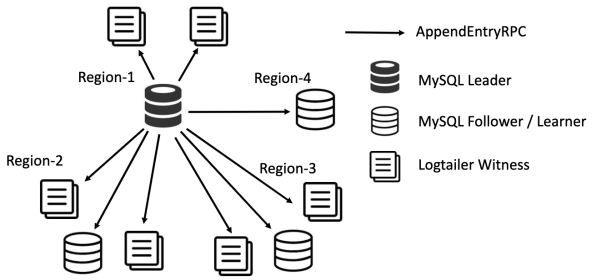
These replication protocols do not provide support for control plane operations, such as electing a new primary or managing membership changes. These operations were handled by external processes (henceforth called *automation*). Automation maintained the notion of a replicaset and handled everything from membership changes to failure detection and primary election. When current primary MySQL was not connectable, automation would orchestrate failover and promote another MySQL to primary. However, the previous "presumed-dead" MySQL could come back to accept writes, where we would have a split-brain. To tackle this issue, we had a mechanism called "node fencing", in which automation disabled writes on the two logtailers of the old primary. This needed additional safeguards like handling restarts, quarantines and complex persistent locking, often leading to corner-case. These automation processes would run on different machines and used external locking to avoid race conditions, and the order of operations would be critical for correctness. The distributed setup made it difficult to accomplish this atomically. Automation grew increasingly complex and difficult to maintain as the number of corner cases that needed to be addressed increased over time, making debugging issues a challenge.

As we dealt with these issues, it became obvious that we needed to bring the source of truth of replicaset membership within the MySQL server, and serialize the data and control operations in the same replicated log. Consensus algorithms have been utilized in both industry and academia to drive the replicated log, and distributed databases are usually built as a state machine over transaction logs. Raft [20] is a consensus algorithm known for its simplicity and ease of understanding. Additionally, Raft is also a complete specification to build a replicated log, with strong leader semantics. Raft provides algorithms for how to replicate data, elect a leader and manage membership. Raft's popularity grew significantly and it became the most popular consensus algorithm in industry, powering many datastores like CockroachDB [4], MongoDB [41] and Yugabyte [11]. Due to its properties of simplicity, understandability and completeness, we chose Raft as the consensus algorithm to build MyRaft.

*This work was conducted while Vinaykumar, Bartlomiej, Yash, Ritwik and Pushap were employed at Meta Platforms



(a) Prior topology of a replicaset



(b) Raft topology of a replicaset without Flexiraft (§4.1) and Proxy (§4.2)

Figure 1: Replicaset topology

1.2 Contribution

In this paper we present several notable contributions. Firstly, we have successfully integrated the Raft consensus protocol into MySQL. This is a complex undertaking since tenured databases like MySQL have many entrenched requirements and behaviors, which do not necessarily line up with the clean separation of concerns necessary for the commonly understood design of a replicated state machine. Often, existing implementations have a very tight coupling of the state machine (database) and the replicated log. We have successfully preserved most of MySQL's external behavior with no impact to our services, and we have brought the source of truth for leadership and membership management into the MySQL server. We have achieved this integration by making multiple changes to MySQL: (1) We built an abstract layer to manage the well known MySQL binary log [21], which is used as the replicated log in Raft; (2) We implemented various hooks for MySQL to interact with Raft during a transaction's commit which enables us to commit a transaction only after it has received consensus in Raft; (3) Whenever promotion happens inside Raft, it orchestrates a sequence of steps through well defined callbacks to transition MySQL primary to replica or replica to primary. The above contributions can serve as a guide to build replicated state machines on top of Raft. The set of orchestration steps that helps automatic failover and membership changes can be easily repurposed for any other replicated state machine or legacy RDBMS.

As our MySQL replicas are distributed across multiple geographical regions, the standard Raft setup suffers from high commit latency due to cross-region network latency. To solve this, we also made important contributions to Raft by implementing FlexiRaft [39], an algorithm for flexible commit quorums. With FlexiRaft's single region dynamic mode, a data commit quorum only consists of the majority of the entities inside a single region, by which we achieve low commit latencies. We also implemented Proxying which reduces cross regional network bandwidth by supporting a hierarchical tree network topology over a logical star topology. Our evaluation shows that MyRaft has reduced failover time by 24x while still achieving similar low commit latencies, compared to the previous setup.

Lastly, we share our operational experience and lessons learned while rolling out MyRaft to our fleet.

The rest of the paper describes important aspects of this project. In §2, we give an overview of how Raft is overlaid on a MySQL replicaset. In §3, we describe changes that we had to make in MySQL server[1] to integrate it with Raft. In §4, we describe improvements and contributions we made to *kuduraft*[2], the open source C++ Raft implementation which was extracted from Apache Kudu [3]. Rolling out this big change needed careful planning, testing and a staged release. In §5, we describe the steps we took to rollout the new system safely and with minimal downtime for our customers. In §6 we present key results from our production deployment. We end the paper with listing some related work in §7 and conclusion in §8.

2 RAFT TOPOLOGY AND MAIN OPERATIONS

2.1 MyRaft Topology

As shown in Figure 1b, in MyRaft, all members of a replicaset including logtailers are members of the Raft ring. Raft ring members are either *voters* or *non-voters*. Voters participate in leader elections by voting, while non-voters are passive entities that receive logs but do not participate in elections. Replicaset members have the following roles in Raft terminology, which can dynamically change: the primary replica is the *leader*, primary capable replicas are *followers*, non-voting replicas are *learners*, and logtailers are *witnesses*. Learners resemble followers, but they do not participate in the voting process and cannot become a leader. All logtailers are voters, but don't have a storage engine. This mapping is summarized in Table 1.

2.2 Main Operations in Raft

The key operations of Raft are data replication, graceful transfer of leadership or promotion, failover and membership changes.

Data Replication: The client write goes to a primary which is the Raft leader. The leader writes the transaction entry to its local log and replicates the log entry by sending *AppendEntries* RPCs to all members of the Raft ring. The transaction is considered "consensus committed" after majority of the voters acknowledge the log entry. It is then committed in the storage engine.

Promotion: During software upgrades and other maintenance operations, the current leader gracefully transfers leadership (*TransferLeadership*) to one of the followers by first making sure that the follower has all of the log entries and then triggering an election on the follower.

Failover: The leader periodically sends heartbeats to all followers. A follower automatically detects a dead leader due to missed heartbeats, and runs elections to become the new leader.

[1] https://github.com/facebook/mysql-5.6
[2] https://github.com/facebook/kuduraft

MyRaft Role	Entity	MyRaft State Machine / Database Role	MyRaft Voter	Prior Setup Role	w/ In-region Logtailers	Database	Read	Write
Leader	MySQL	Primary	Yes	Primary	Yes	Yes	Yes	Yes
Follower	MySQL	Failover replica	Yes	Replica	Yes	Yes	Yes	No
Learner	MySQL	Non-failover replica	No	Replica	No	Yes	Yes	No
Witness	Logtailer	N/A	Yes	Semi-Sync Acker	N/A	No	No	No

Table 1: Roles in MyRaft compared to prior setup

A logtailer can also be elected as a temporary leader due to Raft's voting rules where the longest log wins. In that case, the logtailer will *TransferLeadership* to a replica using a regular promotion. During the failover process, the erstwhile leader is fenced off by Raft's term increment, and demotes to follower once it is able to communicate with the replicaset again (by receiving *AppendEntries* from the new leader or *RequestVote* RPC from a candidate during voting).

Membership Changes: Membership changes are always initiated by automation. Automation may detect that a member needs to be replaced for a variety of reasons, e.g. load balancing, failure, maintenance, etc. It allocates and prepares a new member and invokes an *AddMember* operation in Raft. The membership change is deemed successful when committed in the log and the new membership information is maintained in Raft. However as per the Raft specification, each individual member of the replicaset uses the new membership list as soon as it writes the change entry to its log. The principles of quorum intersection guarantee the safety property. Quorum intersection is implicitly achieved by allowing only one membership change at a time (a single *AddMember* or a single *RemoveMember*).

3 INTEGRATION WITH MYSQL

We chose Apache Kudu [3] instead of implementing the Raft protocol ourselves because it is an industry proven, production hardened C++ Raft implementation. Kudu has its own replicated log format, while MySQL uses Binary Logs [21] for replication. We have internal tools and services that depend on the binary logs like our backup and restore service and downstream services for change data capture (CDC) [37]. To preserve our MySQL server's external behavior as much as possible and to reduce the burden of migrating to a new log format we decided to support binary logs in MyRaft. Furthermore this decision would make MyRaft simpler for rest of the MySQL community to use. MySQL commands like `SHOW BINARY LOGS` [30], `SHOW MASTER STATUS` [31], `SHOW REPLICA STATUS` [32], `PURGE LOGS TO` [25] and `FLUSH BINARY LOGS` [24] continue to work in MyRaft. Some replication commands like `CHANGE MASTER TO` [23], `RESET MASTER` [28] and `RESET REPLICA` [29] were adjusted or disallowed because these operations are handled by Raft. We also decided to preserve Global Transaction Identifiers (GTIDs) [27] and all other metadata associated with them (like GTID sets). Additionally, every transaction is assigned an *OpID* (Raft term and log index) by Raft.

In this section, we first describe the overall architecture of MyRaft – Raft as a MySQL plugin in §3.1, followed by MySQL relay-log and binlog (§3.2) and MySQL state change during Raft role change (§3.3). We also explain how commit paths work in MyRaft for the leader (§3.4) and follower (§3.5). More details about Raft integration with MySQL log purging (§A.1) and MySQL recovery (§A.2) are included in the appendix.

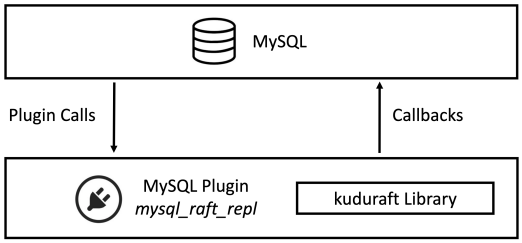


Figure 2: Interaction between MySQL server, plugin and kuduraft

3.1 *mysql_raft_repl*: Raft as a MySQL Plugin

We implemented the state machine/database and Raft interaction through a new MySQL plugin called *mysql_raft_repl*. The plugin uses *kuduraft* as its Raft library. MySQL interfaces with the plugin through the MySQL Plugin API (Figure 2). Similar APIs had been used previously to implement the semi-synchronous protocol [36] as well. For callbacks (Raft calling back into MySQL), we created a separate API. The callback API from Raft to MySQL server is used by Raft to orchestrate a set of steps to configure MySQL as a primary (and then enabling client writes on it) on promotion, and to configure the MySQL to replica (by starting the applier thread and disabling client writes) on demotion. The API is generic and other RDBMS systems can follow the design or specialize their own handlers which would allow them to build a Raft-driven replicated state machine.

Apart from the functionality of orchestrating promotion steps, the *mysql_raft_repl* plugin also provides a helper for *kuduraft* to read and write MySQL binary logs. Since *kuduraft* is a generic Raft implementation, it is unable to natively read MySQL's binary log files. Hence we enhanced *kuduraft* to have a log abstraction layer, and then specialized this abstraction for MySQL in the plugin. This abstraction enables *kuduraft* to read and write transactions from binary logs without having to worry about its format. One instance where the log abstraction is utilized is as follows: Consider the situation when a Raft follower has significantly fallen behind the leader, such that the transaction it needs to be sent are not present in the in-memory cache of the Raft leader. In this case the Raft leader uses the log abstraction to parse historical binary log files, present on the local disk, and then invokes *AppendEntries* RPCs to send these transactions to the follower.

3.2 Relay-Log and Binlog Modes

A MySQL instance has two kinds of replication log files, *binlogs* and *relay-logs* [34]. During asynchronous MySQL primary-replica replication, the MySQL primary writes to its binlog using a pipeline of stages (described in §3.4) and then ships the transactions to replicas (described in §3.5). A replica receives the transactions in relay-logs and executes them using special

threads [26] (henceforth called the *applier* or *applier threads*) that use the same pipeline of stages to commit the transactions to the database.

A MySQL primary uses the Raft plugin to write the transactions into its binlog. The Raft leader also caches this transaction into its in-memory cache and then initiates the *AppendEntries* RPC to the followers. When the Raft follower receives the *AppendEntries* RPC, it unpacks the RPC payload and writes the transaction to the Raft log, which is pointing to the relay-log of MySQL. The write of a transaction to the log file always happens via the Raft plugin. On the primary, MySQL uses the Raft plugin to write the transaction to log file, while on follower the Raft plugin writes the transaction to the log first and then informs MySQL of the new entry. For this mechanism to work, the current log file is kept up to date in Raft by MySQL. During a promotion, an important part of the promotion orchestration process is to rewire these registered files to match the correct persona of MySQL (binlog or relay-log). As mentioned previously, these personas are merely of relevance to the MySQL codebase, which uses different replication logs during different parts of replication (applier vs. the primary).

3.3 MySQL state change during Raft role change

In a healthy Raft ring, the leader continues to replicate writes to all the other members in the ring. In order to preserve authority on the ring, the leader also sends heartbeats if there are no writes. Failovers are initiated within Raft and can happen at any point in time. Process crashes, host crashes, and networking failures are common reasons for triggering a failover. Raft failovers happen when successive heartbeat failures are detected, after which a follower transitions to candidate and starts an election. During a successful failover, Raft will undergo successful role transition first. At this point the Raft role (leader or follower) of the member will not match with the MySQL role (primary or replica). MySQL will then have to be configured by Raft via the callbacks: the state machine/database will be directed by Raft to switch into the desired role, primary or replica.

Before getting elected, a new leader was a follower and its MySQL was behaving like a replica. On transition to leader, the Raft plugin orchestrates the transition of MySQL from a replica to a primary via the following steps:

(1) Appending a No-Op event to the log to assert leadership on the cluster and consensus-commit the tail of the log.
(2) Catching up and committing all transactions to the engine up to the No-Op event using the applier.
(3) Rewiring the logs to switch from *relay-log* mode to *binlog* mode.
(4) Allowing client writes on the primary.
(5) Updating the service discovery system about the change of role to primary.

On the other side, the previous Raft leader, now a follower, must be demoted to a MySQL replica. When the erstwhile leader contacts the rest of the ring (or votes for a candidate at higher term), it notices a change in term. At that point, Raft demotes the member from a leader to a follower. Then Raft initiates the set of orchestration steps to demote MySQL from primary to replica. These steps involve:

(1) Aborting all in flight transactions (transactions that were waiting for consensus commit). Since these transactions are in prepared state in the storage engine, we can simply perform a rollback online.
(2) Disabling client writes on the replica.
(3) Rewiring the logs on MySQL from *binlog* to *relay-log* mode.
(4) In some conditions, there are pending transactions that have not been consensus committed, Raft could truncate the replicated log to remove transactions there were not deemed consensus-committed, and through appropriate callbacks, the GTIDs of the truncated transactions would be removed from all GTID metadata.
(5) Starting the applier thread and pointing it to the appropriate starting point on the relay-log based on an online recovery protocol, where the last transaction committed in engine would be used to determine the starting cursor for the applier.

3.4 Write Transaction on MySQL Primary

Figure 3 demonstrates at a high level the commit paths in MyRaft for the leader and follower. Clients send their writes to a MySQL primary. In a stable ring, the MySQL primary is also the Raft leader. The transaction would first be prepared in the storage engine (e.g. MyRocks or InnoDB). This would happen in the thread of the client's connection. The act of preparing the transaction would involve interactions with the storage engine (e.g. writing prepare markers in the storage engine write ahead log). The thread would acquire appropriate locks based on the isolation level, and the writes would generate an in-memory binary log payload for the transaction. The existing deployments run row based replication (RBR) [35]. The transaction payload created would be according to the mode [22] currently configured, e.g. all the columns in the before-image of the row and the after-image of the row in the database could be part of it. At the time of commit, GTIDs would be assigned, and then Raft would assign an *OpId* (term and index) to the log entry. A checksum is generated for the transaction at this point, to detect corruptions later. Then Raft compresses the transaction and stores it in its in-memory cache and writes the uncompressed transaction to its log file. Once the leader has written the transaction to its own log, the self-vote for the log entry is given. Asynchronously Raft starts shipping the transaction to other followers to get acknowledgements/votes and reach "consensus commit".

The thread that is committing a transaction has three stages in the pipeline. Each stage has its own mutex, and the set of transactions that are grouped together move down each of the three stages of the pipeline in tandem.

(1) **Flush**: The transactions are logged to the binlog via Raft.
(2) **Wait for Raft Consensus Commit**: The thread is blocked waiting for consensus commit on the last transaction in the group. Raft reaches consensus commit when it gets acknowledgements from a majority of the voters. Reaching consensus commit unblocks the thread and it reaches the final/third stage for this group.
(3) **Storage Engine Commit**: In this stage the prepared transactions are durably committed to the engine. After engine commit, the write query will finish and return success to the client. The locks held by the transaction are released at this point, allowing for subsequent, blocked transactions to proceed.

On the Raft side, once "consensus commit" has been reached, the commit marker is moved forward. Raft will piggyback the

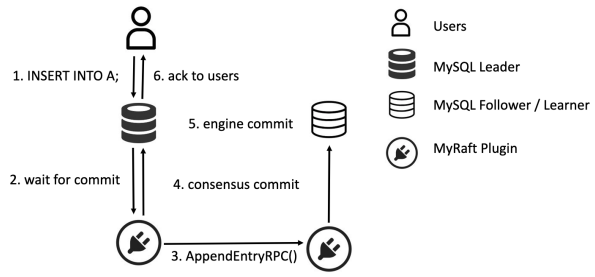


Figure 3: MyRaft Commit Path

commit marker (*Opid* of current commit) to followers in the next *AppendEntries* RPC, so that they can also apply the transactions to their database by proceeding in their applier threads.

3.5 Write Transaction on MySQL Replica

Upon receiving an *AppendEntries* RPC, a non-*leader* will append the transaction to the log using the plugin and the log abstraction. This log file points to the relay-log file of the MySQL replica. Once the new transaction has been appended, the Raft plugin informs MySQL of the new log entry. It also signals the applier thread, which picks up the transaction from the relay-log and applies the transaction to the database. One notable difference is that the applier is processing a binary log payload of RBR events, and is much more efficient in applying the transaction. The last event in a transaction payload is the *COMMIT* event. This triggers the transaction to reach commit and it enters the commit pipeline (similar to the primary). The commit pipeline on the replica goes through the same three stages as a primary; the first stage is the flush stage, where the transaction is written to an applier's log files; the second stage is the wait for consensus commit stage, where it waits for Raft to reach that commit marker; and the final stage is the engine commit phase, where the transaction gets committed to the engine. The deployment does not support chain replication and there are no more downstream Raft entities. Hence, the applier's log is a non-replicated local log. One important design decision we made was to preserve symmetry between the primary and the replica, and therefore the transaction waits for consensus commit by consulting Raft in the same way on both the primary and the replica. On the leader, consensus commit is observed when the quorum votes have arrived, while on the follower, consensus commit is observed when the commit marker sent by the leader arrives.

4 IMPROVEMENTS TO RAFT

4.1 FlexiRaft

The Raft algorithm does not offer much flexibility in choosing data commit quorums, which is important for latency-sensitive applications. Both the leader election and data commit quorums are defined as the majority of the voting members in the replicaset. Even though this configuration provides algorithm simplicity, it is unable to serve production workloads that require lower latency and high throughput. A replicaset in our deployment might have numerous replicas spread out across multiple geographical regions. Seeking a majority vote from members situated in multiple geographical regions was prohibitive and easily became a bottleneck for our workloads. Therefore, we added support for configuring flexible quorums with Raft based on the needs of the application.

Quorums in FlexiRaft are defined in terms of majorities within disjoint groups of members in a replicaset. These groups are constructed based on physical proximity, e.g. a geographical region. If the data commit quorum was defined [14] as a majority in *any* one of the geographical regions spanned by the replicaset, the corresponding leader election quorum would *always* require a majority vote from *every* region, since the tail of the log could've been present in any of those constituent regions.

This is not desirable from a fault tolerance perspective because the disruption of any single geographical region would cause leader elections to fail even if that region did not host any former leaders.

The single region dynamic mode offered by FlexiRaft solves the problem stated above by restricting both the data commit and leader election quorums to a single region and is able to offer latencies on the order of hundreds of microseconds. The implementation maintains region-based watermarks (tracking which log entries have been received by each member), allowing the leader to reach consensus commit on a log entry as soon as acknowledgements have been received from its in-region data quorum (a self-vote from the leader and an acknowledgement from one of the two in-region logtailers). As the name suggests, each successful leader election leads to the data quorum shifting to the next leader's region dynamically. The desired quorum intersection is achieved by keeping track of the last known leader and voting history on each server. More details about the modified algorithm are available in the FlexiRaft paper [39]. The election safety, state machine safety, and leader completeness properties are guaranteed similar to Raft [20] mandating the presence of all previously committed entries in every future leader's log. The correctness of the protocol has been validated by a TLA+ specification [40].

In the above mode, there is a possibility of a rare event of the leader's full region suffering a network partition. The situation rarely happens in production deployments due to power path and networking redundancies in our datacenters. FlexiRaft allows applications to chose their consistency vs latency tradeoffs, and provides them with configurations to have single-region or multi-region commit quorums. Our current deployments however primarily use the single-region commit quorum mode. Since *kuduraft* does not implement automatic step down, the tail of consensus committed entries might increase. In such a situation we currently choose consistency over availability and wait for the network partition to heal. [mike: is this always true? haven't we taken data loss to get availability in bad cases?]

Another outcome of in-region quorums is that during leader failure, the in-region logtailer could be elected as an intermediate leader due to the leader completeness property of Raft (the logtailer tends to be the most-ahead member with the longest log). In such a case, a graceful *TransferLeadership* is utilized to elect a MySQL server as the new leader in a different region.

4.2 Raft Proxying

One requirements mismatch between the standard Raft protocol and the pre-existing multi-region replication strategy we had is the *hub-and-spoke* network architecture of Raft; The leader node replicates data to each follower node directly. Since we have multiple entities distributed across multiple regions in our replication architecture (due to the presence of logtailers), to minimize cross-region data transfer, and to prevent the leader

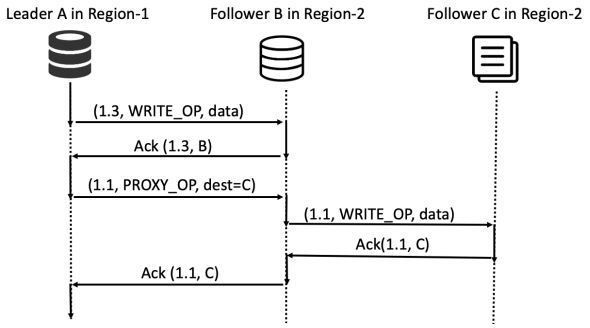


Figure 4: Communication between Leader, Follower as Proxy, and Follower as Destination

from becoming a hotspot [10], we extended Raft's design to support Proxying.

4.2.1 Design. We did not want to make significant logical protocol changes for network optimization purposes, due to the burden of proving the safety of those changes, as well as to keep the changes to the original *kuduraft* implementation smaller. Our design for Proxying keeps all replica log bookkeeping in the leader, making it effectively standard Raft from a safety perspective.

We have added the ability for followers to *proxy AppendEntries* requests from the leader to downstream nodes. In addition, Raft *AppendEntries* messages were made to support a new type of message called `PROXY_OP`, which carries request metadata but no payload. At the final hop, when the message is about to be delivered to the destination node, the final proxy node will reconstitute the payload from its own log, replacing the `PROXY_OP` with the actual log entry represented by the term and index (OpID) specified in the message header (Figure 4).

If the proxy node does not have a local copy of the given entry, it will wait for a configurable period for that message to be available in its local log entry cache before degrading the proxied replication message to a simple heartbeat to the downstream follower, instead of the actual data. Either way, the response from the downstream follower will then be proxied back upstream, ultimately to the leader.

Due to the above blocking / waiting behavior, the leader does not have to worry about specially ordering the transmission of messages containing proxy requests and messages containing a data payload. This optimizes for the common case where all nodes are essentially caught up to the leader. Without the waiting behavior, that case could easily degenerate to 2 or more RTTs across the WAN to replicate ops from leader to remote logtailer.

Note that proxying only applies to data path replication, not votes. Leader election voting continues to operate in a peer-to-peer fashion, and is not proxied. Voting is a rare enough occurrence that the additional complexity is not justified for the cross-region bandwidth savings.

4.2.2 Tradeoffs. One downside to this design is that there is network overhead for control plane requests continuing to go from the leader to all global followers. Our back-of-the-envelope calculation says that proxying to a remote logtailer with the above simple implementation of `PROXY_OP`s is 2-5% of the resource burden of "vanilla" Raft on a per-connection basis, assuming an average of 500 bytes of data per log entry. We believe that the simplicity of the approach and compatibility with the Raft protocol are worth this small overhead.

4.2.3 Failure handling. Sometimes, proxy nodes become unavailable. When this happens, the upstream node (typically the leader) eventually detects their unavailability via health checks and "routes around" them to directly access the next hop, until they become healthy again or are replaced by the control plane infrastructure.

4.3 Mock Elections

[mike: I rewrote this; can someone review it?]

Leader elections are disruptive and can cause write unavailability because leaders have to be quiesced (put into a read-only mode) every time a leader election occurs. Raft pre-elections are intended to mitigate this disruption, but are not sufficient for *TransferLeadership* with FlexiRaft. A graceful *TransferLeadership* in *kuduraft* does not run a Raft pre-election: the only criteria for the election to start on the intended future leader is that it be fully caught up to the current leader. *FlexiRaft* needs a region-based commit quorum, so for a candidate to be elected it also needs a majority of votes from its own region. At least two classes of issues can create write unavailability in this scenario: (1) lagging in-region logtailers can prevent a new leader from committing any transactions until they "catch up" to the commit marker; (2) server bugs can prevent a new leader from completing the promotion process within MySQL. Since *kuduraft* does not implement auto step down, a "stuck" leader can cause problems for a long time.

In order to address these write availability issues, we added an additional election stage called *mock elections*. A mock election round is run before *TransferLeadership* begins, so clients don't see any downtime (user writes are not disallowed yet). The leader captures a snapshot of its cursor and asks the remote member to run a round of pre-election with this snapshot. This mimics the act of queiescing the leader. The rules of voting were modified to reject votes when a member is lagging in the same region as candidate. In essence, the mock election acts as a simulated precheck to confirm that the remote member, with the support of its region's quorum, can win the election and properly function as a leader afterward. This feature has eliminated situations of availability loss, which were often caused by unhealthy logtailers not being replaced quickly enough in the replicaset.

5 SAFE ROLLOUT OF RAFT

As MySQL is a critical service at Meta, it was necessary to maintain the availability and correctness of MySQL during Raft rollout. In this section, we will discuss how we successfully rolled out Raft without disruption to services.

5.1 Shadow Testing

We used a testing tool called MyShadow [19], which generates a production-representative workload, and allows us to test services in an isolated environment. In such a testing environment, we leveraged two types of testing to reveal bugs and inefficiencies early on. The first type of testing was failure injection testing, where we constantly insert crashes into the current leader (triggering failover). The second type of testing was functional testing, where we constantly asked the current leader to gracefully transfer leadership to another member, as well as constantly triggered membership changes. For each type of testing, we repeated the same test thousands times, during which latent bugs started to

reveal themselves. We also did similar crash recovery tests for followers, learners, and witnesses.

During testing, we continuously ran checks to ensure the correctness of the underlying MySQL database and MySQL binary log. The checks were conducted in the form of checksum comparisons between the leader and followers, which is beyond the scope of this publication. In addition, during the testing, we also measured client-side downtime. The overall goal of such testing is to exercise the critical code paths in a production-like environment. We fixed any bugs or system design issues that would prolong the downtime or cause correctness issues.

Lastly, in order to ensure that the downstream dependencies were well tested, we included our change data capture (CDC) and backup/recovery systems as part of the testing.

5.2 Rollout

For production rollout, we created a tool called *enable-raft* to orchestrate the transition from semi-sync to Raft. The tool consists of several steps:

(1) Holding a distributed lock for the replicaset to prevent other control plane operations from executing on the replicaset.
(2) Performing safety checks to determine whether the replicaset is suitable target. For example, we prevent enable-raft from running on a replicaset that is undergoing maintenance.
(3) Loading the plugin and setting the Raft-specific configurations on each of the entities (MySQL and logtailer instances).
(4) Stopping all writes from the client, making sure that all replicas are caught up and consistent, and starting the Raft bootstrap.
(5) Publishing the change of role to primary, as applicable, to the service discovery system.

Once a leader is elected in Raft, the orchestration steps of Raft ensure that client writes are enabled for the leader (§3.3). The last step involves a small amount of write unavailability for the replicaset (usually a few seconds).

The *enable-raft* tool was made robust over time and can roll out Raft at scale in a very short period of time. We have used it to safely roll out Raft to thousands of replicasets per day.

5.3 Quorum Fixer

As discussed in the previous sections, to optimize commit latency, the data commit quorum in our FlexiRaft setup is usually small (in the form of one MySQL and two logtailers). A "shattered quorum" (loss of quorum) happens when a majority of the entities in the data commit quorum (two out of three entities) are unhealthy, which can cause a loss of write availability. Although the software has been well-tested and hardened for production, failures can still happen when running services at scale. The typical case is when automation does not detect unhealthy instances/logtailers in the ring and does not replace them quickly enough. This can happen due to poor detection, worker queue overload, or a lack of spare host capacity. A rarer case is correlated failures when multiple entities in the quorum go down at the same time. This does not happen often, because the deployments try to isolate failure domains (such as power and network) across critical entities of the quorum, through careful placement decisions.

When incidents happen, tools need to be available to mitigate such situations in production. We anticipated this, and hence built *Quorum Fixer*. Quorum Fixer is a remediation tool authored in Python which helps to restore the availability of the Raft ring in the case of a shattered quorum. It operates as follows: (1) First, the tool quiesces attempted writes on the Raft ring. (2) Next, it performs out-of-band checks to determine which entity has the longest log, therefore should be chosen as the next healthy leader. (3) Then, it forcibly changes the quorum expectations for a leader election inside Raft, so that the chosen entity can become a leader, despite not winning enough votes. (4) Finally, after a successful promotion, it resets the quorum expectations back to normal, and the ring becomes healthy.

By default, Quorum Fixer runs in a safe, conservative mode, but has configuration options to relax various safety constraints. Most of the time, running it in its default conservative mode is sufficient to restore write availability to a replicaset. [mike: I added the previous two sentences, looking for a sanity check on accuracy.] Although Quorum Fixer is an important tool for mitigating problems in production, we made the decision to not run this tool automatically. The main reason for this is that we want to identify and root-cause all cases of shattered quorums and eliminate bugs along the way, rather than have them be fixed by automation.

6 EVALUATION

This evaluation focuses on a performance comparison between MySQL's semi-synchronous / asynchronous replication (prior setup) and MyRaft, and shows that MyRaft reduces failover downtime by tenfold while maintaining similar commit latency and throughput. The scalability of the MySQL deployment or comparison with other Raft implementations is not in the scope of this paper.

6.1 Latency and Throughput

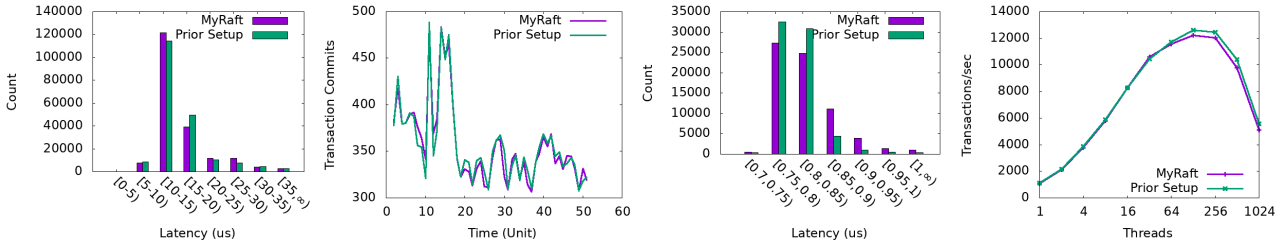
In our production deployment, clients did not notice any regression in commit latency and throughput. However, when pushed to the limits we expected our prior setup to perform better than MyRaft because Raft is a more complex protocol compared to MySQL's semi-synchronous and asynchronous replication.

We used an A/B test with a workload representing production to compare MyRaft with our prior setup. We used a replicaset topology where a primary had two logtailers in the same region, five followers (with two logtailers each) and two learners. All followers and learners were in different geographic regions. The latency between clients and the primary was about 10ms.

Figure 5a shows a histogram of commit latency as observed by the clients. While MyRaft shifts a little towards higher latency, the average latency is very similar: 15758.4us for MyRaft vs. 15626.8us for the prior setup, representing a 0.8% win for the prior setup. Figure 5b compares the throughput between MyRaft and the prior setup with unit time in the x-axis and number of commits in the y-axis. The results showed no significant difference in throughput.

We also evaluated MyRaft using the sysbench OLTP write benchmark [16] which has a much higher write rate compared to our production deployment. To reduce the affect of client to primary latency, we ran the sysbench clients on the same machine as the primary.

Figure 5c shows a histogram of commit latency for the sysbench workload. We observed that MyRaft has a higher latency distribution: average latency was 826.368us for MyRaft vs 811.178us for the prior setup, which is about a 1.9% difference. Figure 5d



(a) Production latency (b) Production throughput (c) Sysbench latency (d) Sysbench throughput

Figure 5: Evaluation for MyRaft vs. Prior Setup

Mode	Operation	pct99	pct95	Median	Avg
Semi-Sync	Failover	180291	98012	55039	59133
Semi-Sync	Promotion	1968	1676	897	956
Raft	Failover	6632	5030	1887	2389
Raft	Promotion	357	322	202	218

Table 2: MyRaft vs. Semi-sync Promotion Downtime (ms)

compares the throughput between MyRaft and the prior setup by number of client threads using the same sysbench benchmark. Up to 32 client threads, we observe similar throughput for both MyRaft and the prior setup. At 128 client threads, both systems reach peak throughput, but MyRaft's throughput is slightly lower by approximately 3%.

In practice, the performance difference between MyRaft and the prior setup is barely noticeable by clients, because most production client traffic is handled by a cache layer called TAO [6].

6.2 Promotion and Failover Downtime

Table 2 shows a promotion downtime comparison between MyRaft and the prior setup. We aggregated this table from production metrics for 30 days. This evaluation demonstrated a marked improvement in promotion and failover times, with manual primary promotions completing within an average of 200 milliseconds; in the previous setup, the average was around 1 second. Dead primary failover was completed in around 2 seconds on average, which includes 1.5 seconds of failure detection time (owing to the 500ms heartbeat configuration and the requirement of three consecutive missed heartbeats to initiate an election). Dead primary failover took 59 seconds in the previous setup. Overall, we saw a 4x improvement in manual promotion time and a 24x improvement in dead primary failover time.

7 RELATED WORK

Raft-based distributed databases: Well-established consensus protocols like Paxos [18] and Raft [20] have become more and more popular among databases. Popular Raft-based database systems include TiDB [15], MongoDB [42], CockroachDB [38], PolarDB [7][8], and Yugabyte [11]. There are two notable differences between these systems and MyRaft: (1) Those database systems are built from scratch with the Raft protocol as a replication layer, while MyRaft was implemented with the goal of integrating with MySQL as a plugin. (2) To the best our knowledge, MyRaft made multiple innovative enhancements to the vanilla Raft consensus protocol that do not exist in previous work. Such improvements include Flexiraft (§4.1), Proxying (§4.2), and mock elections (§4.3). MongoDB [42] revised the Raft protocol to support pull-based data synchronization so that Raft followers could pull data from any Raft follower. MyRaft adopted a different approach by using Proxying to reduce inter-datacenter traffic without modification to the existing Raft consensus protocol.

Highly available MySQL: MySQL's Group Replication [33] solution, which was made available with MySQL version 5.7, has similar end goals as MyRaft. Similar to MyRaft, it aims to make MySQL a true distributed, self-managed database. Group Replication is based on Mencius [5] and was built to primarily support multi-primary setups. Its strong leader and single primary support was not ready when MyRaft started. It also did not support flexible quorums like FlexiRaft, which are required for in-region quorums. Xenon [2] is a MySQL cluster management tool that leverages an out-of-band heartbeat to detect primary failure and conducts automatic failover based on MySQL GTIDs. The idea is similar to the prior setup, where failure detection and failover are orchestrated by automation. In MyRaft, we have integrated the Raft consensus protocol into the MySQL server itself.

8 CONCLUSION

In this paper, we have presented MyRaft, a MySQL database integrated with the Raft consensus protocol. We have made MySQL a natively fault-tolerant database with strong correctness guarantees. In order to meet the workload requirements at Meta, we implemented FlexiRaft to reduce commit latency and Proxying to reduce cross-datacenter traffic. Our production experience has shown that MyRaft reduces failover time by tenfold, while still achieving performance comparable MySQL with semi-sync replication. Retrofitting a consensus algorithm onto a legacy RDBMS revealed a set of challenges that had to be systematically overcome. Careful planning, monitoring, accompanying automation, and mitigations were required to be in place to make the rollout of Raft at scale possible.

ACKNOWLEDGMENTS

This project would not have been possible without the hard work of the MySQL team at Meta. We would especially like to extend our gratitude to Shrikanth Shankar and David Nagle for helping get the project funded and for their insights and guidance, and to Affan Dar and Jim Carrig for supporting the team through the project. We would also like to extend thanks to our technical program managers Dan O and Karthik Chidambaram for keeping us on track during this complex project.

REFERENCES

[1] 2014. *Semi Synchronous Replication at Facebook.* http://yoshinorimatsunobu.blogspot.com/2014/04/semi-synchronous-replication-at-facebook.html

[2] 2017. *Xenon.* https://github.com/radondb/xenon
[3] 2023. *Apache Kudu.* https://github.com/apache/kudu
[4] 2023. *CockroachDb: Replication Layer.* https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html#raft
[5] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*.
[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} Meets Computational Storage: Efficiently Support Analytical Workloads in {Cloud-Native} Relational Database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.
[8] 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.
[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. http://labs.google.com/papers/bigtable.html
[10] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 235–247. https://doi.org/10.1145/3448016.3452834
[11] Sid Choudhury. 2018. *How Does the Raft Consensus-Based Replication Protocol Work in YugabyteDB?* https://www.yugabyte.com/blog/how-does-the-raft-consensus-based-replication-protocol-work-in-yugabyte-db/
[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.
[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. ACM, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450
[14] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. https://doi.org/10.48550/ARXIV.1608.06696
[15] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[16] Alexey Kopytov. 2012. Sysbench manual. *MySQL AB* (2012), 2–3.
[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922
[18] Leslie Lamport. 2001. Paxos Made Simple.
[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] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. http://dblp.uni-trier.de/db/conf/usenix/usenix2014.html#OngaroO14
[21] Oracle. 2023. *The Binary log.* https://dev.mysql.com/doc/refman/8.0/en/binary-log.html
[22] Oracle. 2023. *MySQL 8: Binlog Row Image Setting.* https://dev.mysql.com/doc/refman/5.7/en/replication-options-binary-log.html#sysvar_binlog_row_image
[23] Oracle. 2023. *MySQL 8: Change Master To Statement.* https://dev.mysql.com/doc/refman/8.0/en/change-master-to.html
[24] Oracle. 2023. *MySQL 8: Flush Statement.* https://dev.mysql.com/doc/refman/8.0/en/flush.html
[25] Oracle. 2023. *MySQL 8: Purge Binary Logs Statement.* https://dev.mysql.com/doc/refman/8.0/en/purge-binary-logs.html
[26] Oracle. 2023. *MySQL 8: Replication Threads.* https://dev.mysql.com/doc/refman/8.0/en/replication-implementation-details.html
[27] Oracle. 2023. *MySQL 8: Replication with Global Transaction Identifiers.* https://dev.mysql.com/doc/refman/8.0/en/replication-gtids.html
[28] Oracle. 2023. *MySQL 8: Reset Master Statement.* https://dev.mysql.com/doc/refman/8.0/en/reset-master.html
[29] Oracle. 2023. *MySQL 8: Reset Replica Statement.* https://dev.mysql.com/doc/refman/8.0/en/reset-replica.html
[30] Oracle. 2023. *MySQL 8: Show Binary Logs Statement.* https://dev.mysql.com/doc/refman/8.0/en/show-binary-logs.html
[31] Oracle. 2023. *MySQL 8: Show Master Status Statement.* https://dev.mysql.com/doc/refman/8.0/en/show-master-status.html
[32] Oracle. 2023. *MySQL 8: Show Replica Status Statement.* https://dev.mysql.com/doc/refman/8.0/en/show-slave-status.html
[33] Oracle. 2023. *MySQL Group Replication.* https://dev.mysql.com/doc/refman/8.0/en/group-replication.html
[34] Oracle. 2023. *The Relay Log.* https://dev.mysql.com/doc/refman/5.7/en/replica-logs-relaylog.html
[35] Oracle. 2023. *Row based replication.* https://dev.mysql.com/doc/refman/8.0/en/replication-rbr-usage.html
[36] Oracle. 2023. *Semisynchronous Replication.* https://dev.mysql.com/doc/refman/8.0/en/replication-semisync.html
[37] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, et al. 2015. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 351–366.
[38] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.
[39] Ritwik Yadav and Anirban Rahut. 2023. FlexiRaft: Flexible Quorums with Raft. *The Conference on Innovative Data Systems Research (CIDR)* (2023).
[40] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods: 10th IFIP WG10. 5 Advanced Research Working Conference, CHARME'99 BadHerrenalb, Germany, September 27–29, 1999 Proceedings 10*. Springer, 54–66.
[41] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou
[42] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB.. In *NSDI*. 687–703.

A INTEGRATION WITH MYSQL (EXTRA)

A.1 Rotation and Purging of Logs

MySQL binary log files need to be rotated so that they don't grow too large in size. This is done by calling `FLUSH BINARY LOGS` [24] on the MySQL instance. In MySQL Raft we made the decision that we wanted to keep the Raft replicated log files identical across the replicaset. This was done for simplicity and maintaining the strong invariant of log equality. As such, rotate events generated on the primary are replicated by Raft. On the primary, log rotation is initiated by external automation by calling `FLUSH BINARY LOGS` while monitoring the size of the last binlog file (by executing `SHOW BINARY LOGS` [30] in a monitoring loop). `FLUSH BINARY LOGS` enters the commit pipeline, generates a Rotate event, and uses Raft to get it consensus committed. Rotate events will have an *OpID* stamped by Raft. Once the rotate event reaches consensus commit, the current binlog file is closed and a new file is opened. The previous-GTID-set of the last file is added to the the header of the next file. The index file that maintains the list of log files is also updated with the new file.

On the followers, the rotate event is written to current relay-log on *AppendEntries*, and then the follower stops accepting any new transactions until the commit marker of the rotate event arrives. Once that arrives, Raft calls into MySQL to rotate the relay-log file. Rotation closes the current file, opens a new file, updates the index, and updates the previous-GTID set in the header of the next file. At that point, the log files on both the leader and follower are identical and can be exactly compared. The reason we need to wait for consensus commit before rotation is because there is a system guarantee that log truncation won't happen after consensus commit [mike: something seems wrong with this explanation] and there is no possibility of truncating a file that is already rotated. A rotated file has only consensus committed events.

Purging is the deletion of old binary log files and the removal of them from the log index. This frees up space on the host. Purging is a local decision because every host can have a different

allocation for how much disk it has for binlogs. As such, there is no replicated event generated for purging. In Raft, we have heuristics to prevent purging of files before they are shipped out of region. This prevents purging logs too soon, as severely lagging out-of-region replicas might request older logs from the primary. Raft maintains watermarks per region, so making this decision is easy. MySQL will only purge a binary log file by consulting Raft. Purging removes entries from the index file for logs.

A.2 Recovery

Changes had to be made to crash recovery to make it seamlessly work with Raft. Crashes can happen at any time in the lifetime of a transaction and hence the protocol has to ensure consistency of members. Here are some important insights on how we made it work.

(1) **Transaction was not written to binlog**: In this case the in-memory transaction payload (still in MySQL process memory as an in-memory buffer) would be lost and the prepared transaction in the storage engine would be rolled back on process restart. Since there was no extra uncommitted transaction in the Raft log, no reconciliation with other members must be done.

(2) **Transaction was written to binlog but never reached other members**: On crash recovery, prepared transactions in the storage engine will be rolled back (since the engine had not reached commit). Raft will go through failover and a new leader will be elected. The leader election will happen according to the longest log rules of Raft and the leader election quorum rules of Raft. Per FlexiRaft, the new leader would get votes from the logtailers in the region of the erstwhile leader (the previous leader's data quorum). The new leader will not have this transaction in its log because the transaction never arrived outside the erstwhile leader's process. The new leader will now gain authority by pushing a No-Op event. When the erstwhile leader rejoins the replicaset, the transaction will be truncated from its log. It should be noted that after crash recovery a member always is in follower mode and therefore the truncation happens while it is a replica (relay log persona). When the truncation happens, the GTID sets are adjusted. The truncated GTID is removed and can be reused later (so that there are no holes).

(3) **Transaction was written to binlog and reached the next leader, previous leader crashed before committing to engine**: Similar to 2 above, the prepared transaction in the engine would be rolled back. The erstwhile leader would join the Raft replicaset as a follower. In this case the new leader will have this transaction in its binlog and hence no truncation will happen, since the logs will match. When the commit marker is sent by the new leader, the transaction would be reapplied again from scratch in the applier thread. The erstwhile leader has demoted to follower in Raft and demoted to replica in MySQL, and the plugin has started the applier threads during orchestration of promotion, and has done the proper cursor positioning for the applier thread to continue.