Fault-Tolerant Replication with Pull-Based Consensus in MongoDB

Siyuan Zhou[1], Shuai Mu[2]
[1]*MongoDB Inc.* [2]*Stony Brook University*

Abstract

In this paper, we present the design and implementation of strongly consistent replication in MongoDB. MongoDB provides linearizability and tolerates any minority of failures through a novel consensus protocol that derives from Raft. A major difference between our protocol and vanilla Raft is that MongoDB deploys a unique *pull-based* data synchronization model: a replica pulls new data from another replica. This pull-based data synchronization in MongoDB can be initiated by any replica and can happen between any two replicas, as opposed to vanilla Raft, where new data can only be pushed from the primary to other replicas. This flexible data transmission topology enabled by the pull-based model is strongly desired by our users since it has an edge on performance and monetary cost. This paper describes how this consensus protocol works, how MongoDB integrates it with the rest of the replication system, and the extensions of the replication protocol that support our rich feature set. Our evaluation shows that MongoDB effectively achieved the design goals and can replicate data efficiently and reliably.

1 Introduction

MongoDB is a general purpose, document-based, distributed database. In the last few years, we have been focusing on improving its support for replication, as there has been an increasing demand for stronger fault tolerance. In previous papers we discussed how MongoDB supports tunable consistency [29] and how causal consistency works [34]. In this paper we present the details of how MongoDB provides linearizable [12] replication with fault tolerance.

A common approach to fault-tolerant linearizable replication is through consensus protocols [24, 4]. After studying the popular consensus protocols including Paxos [17] and Raft [26], we concluded that no existing consensus protocols directly fit our needs without heavy modifications. The key reason is that these existing protocols are *push-based*: there is usually a primary server and the primary will push new data to all replicas. Yet in MongoDB we aim for a *pull-based* synchronization model: a replica fetches new data proactively from another replica, and not necessarily from the primary.

There are a few reasons for why we target the pull-based synchronization model. First, allowing data synchronization to happen between any two replicas enables a more flexible data transmission topology, that could utilize networks in more optimal ways. Many of our users prefer being able to configure how their network is utilized, especially for those who deploy their systems across different datacenters. Second, using a pull-based data synchronization model gives us backward-compatibility as we have previously implemented a preliminary version of a pull-based primary-backup replication scheme that is not backed by any consensus protocols, and thus has limited fault tolerance.

As there is no direct fit, we developed a new replication (consensus) scheme based on the Raft protocol. We chose Raft as the base because it is more accepted for industry use, easy to understand, and similar to our previous replication protocol, but we believe other bases such as Paxos should work too. The principle of our approach is to decouple data synchronization in Raft (mostly the AppendEntries RPC) into two parts: replicas pulling new data from the peers, and replicas reporting their latest replication status so that a request can commit after it reaches a majority of replicas.

The development of the new replication scheme is, however, easier said than done. The main challenge is in the subtlety of the Raft (and any other) consensus protocol. During our development we found that any unthoughtful changes to the protocol would easily introduce new corner cases that would break the correctness of the system. To verify that our design and implementation are correct, we have done extensive verification and testing on the protocol including model checking using TLA+, unit testing, integration testing, fuzz testing [11] and fault-injection testing.

Our developed protocol achieves the goal of allowing data pulling between *any* two replicas. Unlike Raft which can only push data from the primary to other replicas (broadcast), our system supports arbitrary data synchronization paths: from linear chaining to broadcast. This gives MongoDB several advantages over using vanilla Raft, including both performance-wise (e.g., saving leader bandwidth) and management-wise (e.g., controlling data transmission paths).

The main contributions of this paper include:

- We design a new consensus protocol based on Raft. Our protocol better meets the needs of MongoDB. It enables more flexible and customizable data synchronization paths during replication.
- We describe the design choices in how MongoDB adopts

this consensus protocol. MongoDB offers a unique feature set, which brings extra challenges in designing and implementing its replication. For example, we provide speculative execution to be compatible with MongoDB's weak consistency features, but also guarantee linearizable log replication with rollbacks.

- We report the evaluation results of MongoDB for different replication parameters. Our evaluation shows that MongoDB replication is reliable and efficient.

We will organize the rest of paper as follows. Section 2 gives an overview of the background. Section 3 describes the main body of the consensus protocol. Section 4 gives a few important extensions to our design. Section 5 discusses the evaluation results. Section 6 discusses related works before we conclude.

2 Background

MongoDB interfaces and architecture. MongoDB is a database that stores data as *documents* and supports general CRUD operations on one or many documents with a rich query language. Each document is a binary JSON-like (called BSON) object. Documents are identified by unique ids and grouped in *collections*, which are similar to tables in a SQL database.

To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB nodes that act as a consensus group, where each node maintains a logical copy of the database state. MongoDB also supports *sharding* for horizontal scaling, which distributes all data in a collection onto different shards in a share-nothing manner. Each shard is deployed as a replica set. In this paper we focus on a single replica set, as sharding is orthogonal.

Consistency and fault tolerance. Previous papers have described how MongoDB can achieve weaker consistency levels, including causal consistency [34, 29]. In this paper, we focus on the strongest consistency level—linearizability [12]. We assume a (partially) asynchronous environment where messages can be arbitrarily delayed and there are no perfect failure detectors. For each replica set, at most a minority of servers can fail in order to maintain availability. The problem of fault-tolerant linearizable replication is commonly solved by consensus protocols. Examples include Viewstamped Replication [24], Paxos [17], Zab [13], and Raft [26]. Our solution started with adopting the recent and popular Raft, but ended up basically inventing a new protocol with many heavy modifications.

Evolution of MongoDB's pull-based replication. Starting from MongoDB version 1.0 over a decade ago, MongoDB supported replication with a primary-backup scheme. Unlike conventional primary-backup replication schemes where updates are usually *pushed* from where they are firstly received—the primary—to the secondaries (backups), we chose a design in which a secondary server can constantly *pull* updates from other servers, and not necessarily from the primary.

A major benefit of the pull-based approach is that it enables a more flexible control of how data is transmitted over the network. Depending on users' needs, the data transmission can be in a star topology, a chaining topology, or a hybrid one. The ability of controlling data transmission paths is a strong customer need, mostly for reasons related to performance and monetary cost. For example, when deployed in clouds like Amazon EC2, data transmission inside a datacenter is free and fast, but is expensive and subject to limited bandwidth across datacenters.

The pull-based approach has led our designs while we continually evolved our replication protocols in the last few releases. In earlier releases several years ago, we assumed a semi-synchronous network: either there is manual control of failover (the user needs to appoint a node as the new primary when the old one fails), or all messages are bounded to arrive within 30 seconds for failure detection. Starting from 2015, we remodeled our replication scheme based on the Raft protocol. This new protocol guarantees safety in an asynchronous network (i.e., messages can be arbitrarily delayed or lost) and supports fully autonomous failure recovery with a smaller failover time. Same as before, this new protocol is still pull-based. We will describe how it works in the next section.

3 Design

This section describes how replication in MongoDB works, including the overall architecture and data structures (§3.1), the main body of the replication protocol (§3.2), a discussion of correctness, (§3.3), and how the system chooses data transmission paths (§3.4). Attached to this paper is an appendix that summarizes the difference between the Raft protocol and MongoDB's consensus protocol.

3.1 Preliminaries

In MongoDB, the object for replication is called the *oplog*. An oplog is a sequence of log entries; each log entry contains a database operation. Figure 1 shows an example of oplog entry. The oplog is stored in the *oplog collection*, which behaves in almost all regards as an ordinary collection of documents. The oplog collection automatically deletes its oldest documents when they are no longer needed and appends new entries at the other end.

An oplog entry needs to be replicated to at least a majority of servers to *commit*: a committed entry persists through any minority failures. The system will wait for the oplog entry to commit before it acknowledges the client. MongoDB also

```
{
  // The oplog entry timestamp
  "ts": Timestamp(1597904287, 12),
  // The term of this entry
  "t":  NumberLong(40),
  // The operation type, "i" for insert
  "op": "i",
  // The collection name
  "ns": "test.collection",
  // A unique collection identifier
  "ui": UUID("947b54f...852f62")),
  // The document to insert
  "o":{
      "_id": ObjectId("5f3e...b950"),
      "x": 1
  }
}
```

Figure 1: Example of key oplog entry fields for an “insert" operation

supports operations with weaker consistency, in which case the system can acknowledge clients before the oplog entry commits [29, 34]. After replication, all servers of the same replica set will have identical oplogs. Oplog entries will be applied in the same order on all servers.

A server can act as either a primary or a secondary. [1] Only a primary can process write requests. A server has a third role as a *candidate* when it is transitioning from a secondary to a primary through *elections*. MongoDB’s election rules are the same as Raft’s. When a node decides to start an election, for example, because it has not seen a primary for an election timeout, the node transitions to a candidate role, increases its term, and sends vote requests to others. A voter can grant its vote to only one candidate in a given term and only if the candidate has the same or a more up-to-date log than the voter. The candidate wins the election if it is able to collect votes from a majority of nodes including the candidate itself; it then becomes a primary.

After the election, the new primary will have a unique, monotonically increasing *term* number. When the primary generates a new oplog entry, it will append this entry into its own oplog, and replicate the entries through the *data replication* protocol described in Section 3.2. It could happen that more than one server is acting as a primary, but the data replication and the election protocols collectively guarantee that at most one primary can successfully commit log entries at a particular index.

In MongoDB, each oplog entry is assigned a timestamp and annotated with the term of the primary. The timestamp is a monotonically increasing logical clock that exists in the system before this work. It is used to index the oplog entries, similar to the log index in Raft. A pair of term and timestamp, referred to as an OpTime, can identify an oplog entry uniquely in a replica set and give a total order of all oplog entries among all replicas. OpTimes are compared lexicographically, i.e., an OpTime is greater than another if its term is higher or the terms are the same but its timestamp is higher.

3.2 Data Replication

As mentioned in previous sections, MongoDB uses a pull-based replication scheme. Unlike Raft and other common consensus protocols that would initiate RPCs from the primary to secondaries when the primary tries to replicate new log entries (for example, in Raft, this is the AppendEntries RPC), in MongoDB, the primary waits for the secondaries to pull the new entries that are to be replicated.

After appending an entry to the oplog, the primary can process two types of RPCs from secondaries: PullEntries and UpdatePosition. A secondary will use PullEntries to fetch new logs, and use UpdatePosition to report its status so that the primary can determine which oplog entries have been safely replicated to a majority of servers and commit them. Similar to Raft, once an entry is committed, all prior entries are committed indirectly.

3.2.1 PullEntries

Note that a key design choice is that a secondary does not have to send PullEntries only to the primary. Instead, the secondary can pull new entries from any (nearby) servers. This secondary is called the *syncing server*, while the upstream server that receives the PullEntries RPC is called the *sync source*.

A secondary continuously sends PullEntries to the selected sync source (see more details about the sync source selection in §3.4) to retrieve new log entries when they become available. The PullEntries RPC includes the latest oplog timestamp (prevLogTimestamp) of the syncing server as an argument. When receiving PullEntries, a server will reply with its oplog entries after and including that timestamp if it has a longer or the same log, or the server could reply with an empty array if its log sequence is shorter. Before returning a response when the log is the same, PullEntries waits for new data for a given timeout (5 seconds by default) to avoid busy looping.

When the syncing server receives the reply of PullEntries, it will try to merge the log entries in the reply into its own oplog. Before merging, it checks if the incoming log entries concatenate with the local oplog. In particular, it checks if the first received entry has the same OpTime as the last local oplog entry. Only if so, the syncing server continues to merge by appending the received entries to the oplog. If the received oplog entries don’t overlap with the local ones and the received entries are "newer" by comparing their last OpTimes, the syncing server will traverse the oplog on its sync source in order to find their last common entry, then

[1] Our choice of terminology is for historical reasons. They are the same as a *leader* and a *follower* used in Raft.

discard any diverged oplog entries since then. Afterwards, the syncing server should be able to pull new data and append it to the local oplog. Discarding diverged logs will require more work than ordinary Raft implementations because of our optimizations on speculative execution (see details in §4.1).

3.2.2 UpdatePosition

After retrieving new entries into its local oplog with PullEntries, the secondary will send UpdatePosition to its sync source, reporting the latest log entry's OpTime. When receiving the UpdatePosition, the server will forward the message to its sync source, and so forth, until the UpdatePosition reaches the primary. The primary keeps a non-persistent map in memory that records the latest known log entry's OpTime on every replica, including its own, as their log positions. When receiving a new UpdatePosition, the primary will compare the received OpTime with its local record. If the received one is newer, the primary will replace its local record with the received one. Afterwards, the primary will do a count on the log positions of all replicas: if a majority of replicas have the same term and the same or greater timestamp, the primary will update its lastCommitted to that OpTime and notify secondaries of the new lastCommitted by piggybacking onto other messages, such as heartbeats and the responses to PullEntries. lastCommitted is also referred to as the *commit point*.

3.2.3 Implementation

In MongoDB, instead of initiating continuous RPC's on the syncing node, the PullEntries RPC is implemented as a query on the oplog collection with a "greater than or equal to" filter on the timestamp field. The query can be optimized easily since the oplog is naturally ordered by timestamp. Using database cursors allows the syncing node to fetch oplog entries in batches and also allows the RPC to work in a streaming manner, so that a sync source can send new data without waiting for a new request, reducing the latency of replication.

To avoid a flood of forwarded UpdatePosition messages, a server passively batches the received UpdatePosition requests between two rounds of forwarding and consolidates the requests by only keeping the highest log position for each server. A server only maintains at most one in-progress UpdatePosition request to its sync source, so it waits to send the next request until the previous one returns.

We also introduced Heartbeats RPC, which decoupled the heartbeat responsibility from Raft's AppendEntries RPC. Heartbeats are sent among all replicas, used for liveness monitoring, commit point propagation and sync source selection (§3.4).

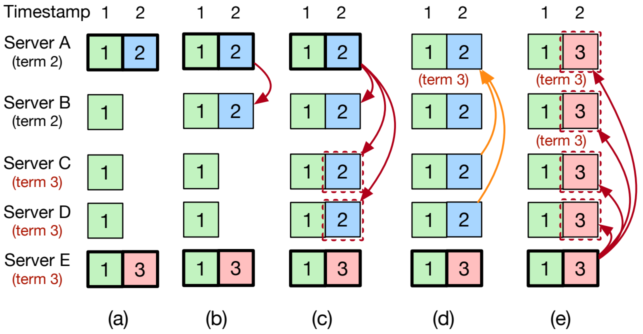

Figure 2: A corner case

Each box represents an oplog entry with its term (shown in different colors) in the box. (a) start state with both A and E being primaries (thick border); (b) Raft only allows replicating (red arrows) from A to B; (c) MongoDB can replicate from A to B/C/D; (d) C/D report log positions (orange arrows) with term 3, forcing A to step down. (e) data replicated in (c) may be rolled back.

3.3 Correctness

Careful readers may have noticed that our data replication protocol (how PullEntries are processed) only checks the OpTimes in the logs; it does not check if the sync source has a higher or equal term than the syncing server. This is different from what Raft would do in data replication. The data replication in Raft is done via AppendEntries RPC, which contains the term of the primary. AppendEntries can only succeed if the primary has a term that is not lower than the secondary's.

This crucial protocol change means that log replication in MongoDB will behave differently from Raft. In Raft, if a server has voted for a higher term in an election, the server cannot take new log entries sent from an old primary with a lower term. But in our system, because the PullEntries RPC does not check the term of the sync source, even if the sync source is a stale primary, it is possible that after a server has voted for a higher term, the server could still fetch new log entries generated by the stale primary.

Figure 2 shows an example of this disparity occurring on secondaries. Initially, all five servers acknowledged entries in term 1. Server A first wins the election in term 2 with votes from Server A/B/C and writes down one entry locally. Server E then wins the election in term 3 with votes from Server C/D/E and writes a different entry locally. In Raft, if Server A broadcasts its AppendEntries, only Server B will accept the new entry from Server A; Server C/D/E will all reject.

In MongoDB, however, Server B/C/D could all take the new log from Server A even after Server C/D have voted for the new primary in term 3. If A is the sync source of Server B/C/D, then B/C/D will still accept the new entry with term 2 because the entry is newer than their local ones. Now that the entry in term 2 has been replicated to the majority of servers, it would be considered by Server A as committed if we did not make further changes to Raft's rule that *a log entry is committed once the leader that created the entry has replicated it on a majority of the servers* [26]. Later, Server E's new entry with term 3 could propagate to all servers and overwrite the committed entry. Without more changes from Raft, this would violate safety. [2]

To prevent cases like this from happening, we added a new argument in the UpdatePosition RPC: the term of the syncing server. The recipient of UpdatePosition will update its local term if the received term is higher. If the recipient is the stale primary, seeing a higher term will make the primary step down before committing anything, thus avoiding any safety issue. In the above example, when server A receives UpdatePosition from Server C/D, it will see term 3 and step down immediately without updating its lastCommitted. Even though the entry with term 2 is in a majority of servers' logs, it is not committed.

Until now, we have assumed it is a secondary that fetches oplog entries from a stale primary after voting for a new primary. In fact, this syncing server could be the new primary itself. Even if a primary (or candidate) has voted for itself in a higher term, it could still fetch data generated in lower terms from other replicas, as long as it has not generated new oplog entries with the new term and appended the entries to its own oplog. This important difference between MongoDB and Raft allows MongoDB to preserve uncommitted data as much as possible during failovers (see more in §4.3).

More formally, the revisions we made to UpdatePosition are to maintain a key invariant in Raft—Leader Completeness Property. This property refers to the fact that "if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms." (See Figure 3 in the Raft paper.) In MongoDB, in order to commit an entry in term T, the primary in term T has to receive UpdatePosition RPCs with term T from a majority of nodes. A later new primary in term U > T must collect votes from a majority of nodes too. The two majorities must overlap on at least one voter. This voter is the key to guarantee the safety. Either the voter sent UpdatePosition in term T to commit the entry **before** voting, thus implying the new primary had the committed entry due to the Log Matching Property; or the voter voted first and sent UpdatePosition with a term higher than T, thus leading the primary in term T to step down without committing the entry. Either way, the Leader Completeness Property is guaranteed.

In addition to this property, other invariants of Raft still hold so that one can prove the correctness of our protocol following the proof of Raft. Further, to mechanically verify the correctness of our system, we have written a formal specification of the protocol in TLA+ and applied model checking to it [33].

3.4 Sync Source Selection

Servers learn about the status of other servers, including their log positions, via Heartbeat RPC. A server chooses its sync source only if the sync source has newer oplog entries than itself by comparing their log positions. This condition is double-checked on receiving PullEntries RPC's responses in case rollback (§4.1) occurs on the sync source. As a result, it's guaranteed that the replicas can never form a cycle of sync sources. Once a server starts to pull entries from its sync source, it keeps fetching from the source until the source is not available or a better source shows up. Thus a server should not change its sync source frequently in a stable environment.

4 Extensions

In this section we introduce a few key features of MongoDB that extend the elementary design.

4.1 Speculative Execution and Rollback

The standard approach of applying log entries in Raft-based systems is that a replica waits until the log entries are committed and then applies the log entries in timestamp order. MongoDB introduces an optimization that speculatively applies an oplog entry when it is added to the oplog. If a failover happens, speculatively applied oplog entries could be deleted (§3.2.1). In this case, the system needs to roll back the operations in these entries. The common approach for rollbacks in databases is through undo or redo logs. The way MongoDB achieves this is through a consolidated design of the storage engine (named WiredTiger) and the replication protocol.

The WiredTiger storage engine is a multi-version transactional storage engine that can use oplog timestamps as versions of data updates. There are three key functions implemented in the storage engine that enable this consolidation. First, the storage engine supports speculative updates, so multiple versions are visible to clients depending on their requested consistency levels even if not all of them are committed. Second, the storage engine provides fast rollback to a timestamp and discards all updates after that timestamp. Third, when oplog entries are committed, the storage engine can be notified to merge all data updates with lower timestamps in the on-disk checkpoint and garbage collect those versions.

When a node needs to roll back, it will determine the newest oplog entry it has in common with its sync source. The timestamp of this oplog entry is referred to as t_{common}. The node needs to truncate all oplog entries with a timestamp

[2] Another angle to look at this problem is that, allowing UpdatePosition from a server with a higher term is actually "counting replicas" rather than "counting replies". Raft has explained well why counting replicas is incorrect in its paper. Indeed, Raft's TLA+ spec in [25] specifies a stronger definition of "commit" than its paper. The definition in TLA+ is similar to our modification: an entry $\langle$*index, term*$\rangle$ is **immediately committed** if it is acknowledged by a quorum (including the leader) during *term*.

after t_{common}. In addition to oplog truncation, it must undo the speculative effects of the operations deleted from the oplog.

Since MongoDB version 4.0, the WiredTiger storage engine has provided the ability to revert the replicated data to the version at a given timestamp. MongoDB periodically informs the storage engine of a *stable timestamp* (t_{stable}), which is the timestamp of the commit point known by this node and must be less than or equal to t_{common} when the node starts rollback.

To undo the effects of truncated oplog entries, the rolling back node reverts its replicated data to the version at t_{stable}, then applies the oplog entries forward from t_{stable} up to and including t_{common}.

4.2 Initial Sync

MongoDB discards stale oplog entries once the storage space used to store the entries reaches a configurable threshold, by default 5% of free disk space at startup. In most consensus systems, such as Chubby and the vanilla Raft, a *snapshot* of the database is obtained before discarding stale log entries. The snapshot will be used by a new server to catch up when joining the system. However, MongoDB does not rely on a snapshot mechanism for this initial synchronization, referred to as *initial sync*.

The major reason for MongoDB not using snapshots for initial sync is that MongoDB has a pluggable storage API that does not require the storage engine to support snapshots. For example, the initial storage engine before WiredTiger at its core used `mmap` [3], which does not support snapshots. This `mmap`-based storage engine needs to be supported due to backward-compatibility.

The initial sync in MongoDB works as follows. First, when a new server is joining, it chooses a sync source and uses this sync source for the whole duration of initial sync. Once the initial sync starts, the syncing node records the current applied oplog timestamp on the sync source as the initial sync start point. Then, the new server starts to clone the database of the sync source by scanning the database. The database scan gets a cursor at the beginning of each collection and iterates over the cursor to the end. Note that this clone may be inconsistent when there are concurrent updates happening in the system—some cloned values are up-to-date, some may be obsolete, and some may be missing. The final step is to fix this inconsistency. The new server will retrieve all oplog entries on the sync source starting from the initial sync starting point, and apply the oplog entries locally on the database. After the database clone, the new server records the current applied oplog timestamp on the sync source again as the initial sync end point. Once the new server applies oplog entries beyond this point, the data becomes consistent and the initial sync is complete. If the sync source fails during the initial sync, the new server chooses a different sync source and restart the process.

Note that some oplog entries may be applied twice in the database on the new server. If an operation gets applied on the sync source after the initial sync begins, the operation's effect may be included in the database that the new server cloned gradually. Later this oplog entry will be applied again on the new server. To avoid any data inconsistency caused by this effect, MongoDB requires any sequences of operations in the oplog entries to be *idempotent*: applying the same sequence of operations multiple times will lead to the same system state.

For operations that change the database states (inserts, updates, deletes, etc.), we changed their semantics during initial sync to make them idempotent. Inserts in MongoDB will be ignored if the document id already exists; updates and deletes will be ignored if the modified document's id does not exist. MongoDB supports rich update operations, such as incrementing a field's value in a document. These operations will be converted to unconditional field assignments by the primary. For example, for an increment request, the primary will read the current value from its local database and compute the result of the increment, and replicate an oplog entry setting the field to the computed result. As a result, the consistency between the new server and its sync source is guaranteed.

4.3 Preserving Uncommitted Oplog Entries

After a failover, the uncommitted oplog entries on the previous primary are likely lost. This would be fine for a different system because the clients could retry uncommitted updates. This is, however, an issue for MongoDB because MongoDB supports fast but weak consistency levels that acknowledge writes as soon as they are applied on the primary or just replicated to a fewer number of nodes than a majority. Thus, a failover could cause a large loss of uncommitted writes. Though the clients are not promised durability with weak consistency levels, we still prefer to preserve their uncommitted writes as much as possible.

For this purpose, we introduced an extra phase for a newly elected primary—the primary catchup phase. The new primary will not accept new writes immediately after winning an election. Instead, it will keep retrieving oplog entries from its sync source until it does not see any newer entries, or a timeout occurs. This timeout is configurable in case users prefer faster failovers to preserving uncommitted oplog entries.

As explained in §3.3, the primary catchup design is only possible because a primary is allowed to keep syncing oplog entries generated by the old primary after voting for a higher term as long as it hasn't written any entry with its new term.

[3] When using this deprecated `mmap` engine, features including speculative execution and rollback are implemented in different approaches and are omitted due to space limitation.

4.4 Additional Replica Roles

4.4.1 Arbiters

MongoDB supports a special replica role called *arbiter*. An arbiter is like a secondary with respect to voting but does not store any data to save the cost of storage and replication while being a tie-breaker in elections. For example, in a Primary-Secondary-Arbiter deployment, when the primary crashes, the secondary can take over with the vote from the arbiter to serve reads and writes. The writes will be applied speculatively but cannot be committed. [4] If a new node needs to be added to the replica set to replace the crashed one through initial sync, the arbiter allows a safe reconfiguration to change the membership.

4.4.2 Non-Voting Members

In addition to fault tolerance requirements, users often deploy replica sets to offload reads from the primary or to access a local data copy with lower latency. For example, users may maintain some replicas for a heavy analytical workload using MongoDB's expressive aggregation framework. These replicas are not deployed for fault tolerance and may have unstable write performance due to the heavy analytical workload. As another example, a geo-distributed application may prefer to read from a nearby datacenter for lower latency, thus need to deploy dozens of replicas globally. Assuming it's extremely rare for more than a few servers to fail at the same time, it would be unnecessary to count all replicas towards a quorum for fault tolerance and undesired to wait for a majority of such many servers to replicate in order to commit writes.

MongoDB introduced *Non-Voting Members* particularly for this purpose. Non-voting members replicate data as normal secondaries, but they do not participate in elections or count towards a quorum for committing oplog entries, as opposite to *Voting Members*. MongoDB supports up to 50 replicas but only up to 7 voting members. Therefore, writes with strong consistency levels can return faster, as long as they are committed after replicating to a quorum of voting members rather than a quorum of all replicas.

Non-voting members work well with the pull-based data replication model since it's possible to minimize their performance impact on the primary and voting members by offloading their significant oplog read workload to other non-voting members as much as possible.

4.5 Election Optimizations

4.5.1 Election Handoff

On failovers, secondaries wait for an election timeout (10 seconds by default) to run for election in order to detect that the primary is no longer available. However, on planned failovers, it is known that the old primary has already stepped down. MongoDB introduced *Election Handoff* to shorten the planned failover time by avoiding the next candidate's waiting.

When a primary server steps down on administrator commands, it will pause new writes and wait for any eligible secondary to catch up its oplog within a user specified timeout. The primary then chooses this caught-up secondary to immediately run for election on a best-effort basis. In common cases, the chosen secondary will win the election and become the new primary. This election handoff mechanism will likely shorten the failover time as it does not need the the election timeout to elapse. This is similar to the leadership transfer extension in Raft.

The election handoff is leveraged by the planned maintenance on MongoDB Atlas[21], MongoDB's hosted database as a service. Atlas uses a rolling upgrade strategy for executing maintenance or infrastructure operations, such as applying security patches, scaling up an Atlas cluster, and upgrading to the latest MongoDB minor versions. As part of the rolling upgrade, the primary will be stepped down and shut down for upgrade. The election handoff minimizes the unavailability window on planned failovers. In fact, the vast majority of failovers (89.03%) on Atlas are caused by planned maintenance (§5.2) and can benefit from the election handoff.

4.5.2 Member Priority

It is common that users have a preference for which server should act as primary, especially in a multi-datacenter setup where users prefer to deploy the primary in the datacenters closest to the applications (clients) for lower latency. MongoDB supports setting election priority among servers in the replica set configuration. If a secondary realizes it has a higher priority than the current primary, it will start an election after a timeout based on its relative priority. The higher the priority it has, the smaller the timeout value will be. In this way, the server with the highest priority is likely to win the election first. If the election fails due to competition but the server still has a higher priority than the new primary, it will continue calling for elections until it becomes primary or the new primary has a higher priority. Setting a server's priority to zero prevents it from running for election.

One potential problem is that when the election caused by a high-priority server fails, it will propagate its larger term number to the rest of the replica set and force the current primary to step down. This disruption is particularly serious when this high-priority server is lagged due to shutdown and rejoins the replica set after a restart. Until this high-priority server is caught up on its oplog, it keeps running elections periodically and causing disruptions. Although other servers will elect a new primary after each disruption, the system will be unavailable for at least an election timeout on every

[4] This speculative execution can benefit reads with weaker consistency levels described in [29].

disruption. To prevent disruptions when a server rejoins the replica set, Raft describes a pre-vote algorithm in Section §9.6 in [25], where a candidate only increments its term and runs the real election if it learns from a majority of nodes that they would grant their votes. MongoDB implements this algorithm with an election dry-run. When a stale candidate with a higher priority starts an election even though it won't be able to win, the candidate will fail during the election dry-run, protecting the existing primary from being disrupted by a higher term.

Note that in rare cases such as network partitions, it could happen that two nodes repeatedly initiate elections and cause liveness issues. This is a different issue from the priority design and having priorities does not make it worse. In these cases, the dry-run mechanism cannot address the issue completely, as liveness in asynchronous networks is impossible to guarantee [9]. In reality we did not find this to be a problem.

4.6 Read-only Operations

A strawman solution to support linearizable reads is to turn read operations into log entries similarly to treating write operations. MongoDB's optimized approach is that if the primary has other concurrent oplog entries to replicate, the primary can piggyback the read linearization point with those entries. Note that this optimization is different from weakly consistent reads although they both skip turning read operations into oplog entries. Even though weakly consistent reads are supported on both primary and secondary nodes and they do not need to wait for synchronization between nodes, linearizable reads can only happen on the primary and need to wait for a roundtrip of synchronization.

5 Evaluation

Our evaluation section has two parts. First, we benchmarked the system under different configurations on Amazon EC2 and report the performance measurements (§5.1). Second, we collected metrics from our own cloud platform, MongoDB Atlas [21], and report the analysis of the operational data on failovers (§5.2).

5.1 Benchmarks on EC2

5.1.1 Setup

Our tests use AWS m5d.2xlarge instances, each with 8 vCPUs, 32GB memory, and a local SSD. We tested 5-way replication with 5 server VMs and 2 client VMs. 3 servers are deployed in the US East (N. Virginia) region and 2 servers in the US West (Oregon) region. The primary is always deployed in the US East region.

We use the following benchmark for our main tests. Each client thread continuously and randomly updates an entire document out of 1 million documents containing one random string field of 1000 bytes in a closed loop. These updates will not return until they are committed. We vary the number of client threads to control the offered load in the system.

To measure the impact of allowing secondaries to sync from other secondaries, we run the tests with two settings: (1) chaining enabled and a secondary in the US West region forced to sync from another secondary in the same region while all other secondaries syncing from the primary, and (2) chaining disabled and all secondaries syncing from the primary, which mimics similar data transmission paths to vanilla Raft.

In addition to the above basic benchmarks, we also tested two interesting cases: (1) a failure recovery test in which we crash the primary, wait for the new primary to step up, and then recover the old primary; (2) a comparison with a previous version of MongoDB that uses a deprecated replication protocol not based on known consensus protocols.

Appendix B has an additional TPCC benchmark.

5.1.2 Benchmark Results

Figure 3 and 4 respectively show the 50-percentile and 90-percentile latency vs. throughput of both chaining enabled and disabled cases. Their performance is almost the same. However, as shown in Figure 5, the cross-datacenter traffic is halved by leveraging chaining, so is its cost. Using \$0.01~\$0.147/GB (EC2's pricing) to estimate, the savings of chaining in this test is about \$300~\$5,000 per month.

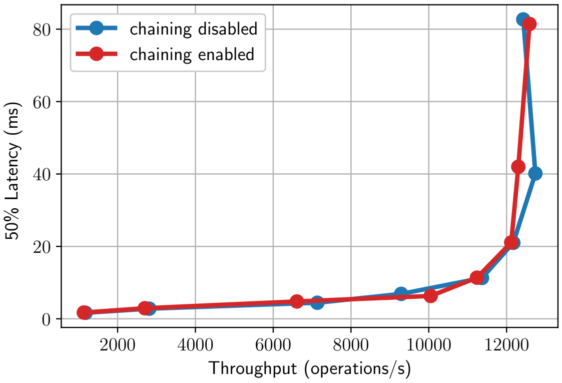

Figure 3: 50-percentile latency vs. throughput of chaining disabled and enabled

One may expect that with chaining enabled the system should be able to achieve a higher maximum throughput because the primary would have more available CPU resources for clients. The system is indeed bottlenecked by the primary's CPU at the maximum throughput in our tests and most of the CPU is used to serve clients. Each client request is handled by a separate OS thread, so there are a few thousand

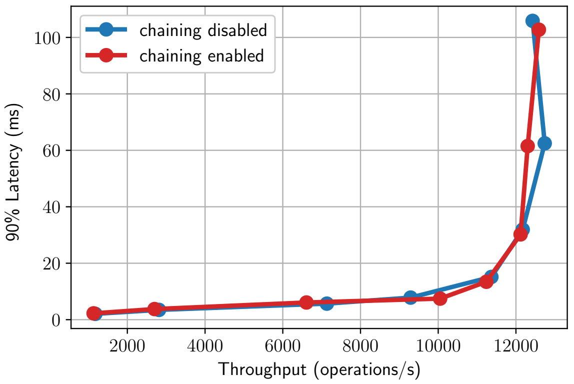

Figure 4: 90-percentile latency vs. throughput of chaining disabled and enabled

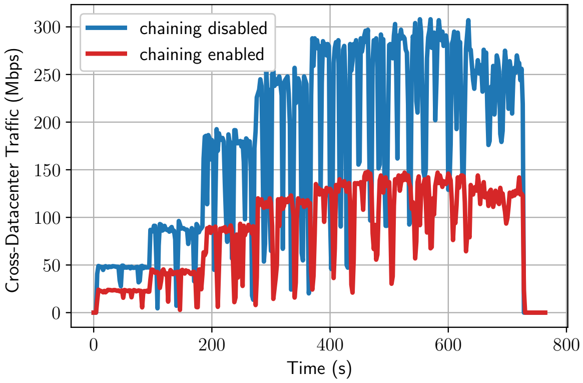

Figure 5: Cross datacenter traffic with chaining disabled and enabled

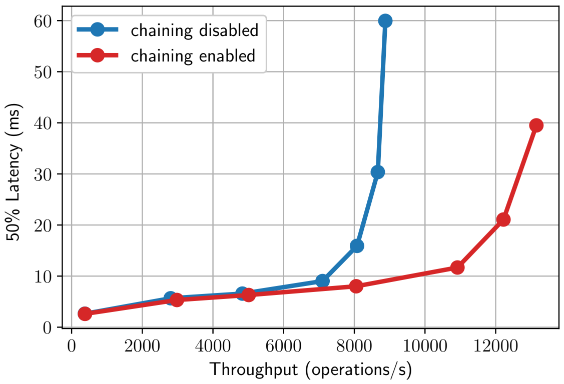

Figure 6: 50-percentile latency vs. throughput of chaining disabled and enabled with limited 200Mbps bandwidth on primary

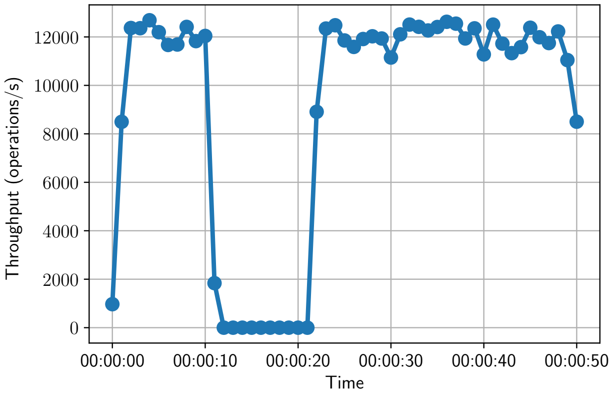

Figure 7: Throughput during failover

client threads in our tests. We observed more than 250k context switches per second on the primary consistently when the system is saturated. The heavy interleaving of threads indicates that the performance will not scale linearly given extra available CPU time. Meanwhile, each secondary oplog read takes one thread on the primary. In fact, a secondary in a steady replication state only consumes about 5% of a single CPU core on the primary in our experiments. Therefore we did not observe throughput improvement with chaining enabled when the primary CPU is the bottleneck.

Nevertheless, if the network bandwidth between nodes is the bottleneck, we expect that enabling chaining will greatly improve the throughput. In our tests, we did not observe throttled bandwidth across datacenters on AWS EC2. But it is reported that the bandwidth of the cloud could be affected by time, space, and other factors such as VM instance types [16]. Besides, our users could deploy MongoDB in other environments with networks that may be less reliable and have less bandwidth than EC2. Therefore, we conducted an additional experiment with 3 nodes, where the primary's bandwidth is limited using the `tc` tool. The result is in Figure 6) and it shows that when the network bandwidth is the bottleneck, chaining can drastically improve the system throughput as expected.

5.1.3 Failover Tests

The failover tests are conducted with all 5 replicas deployed in the same datacenter. In each test, we run 64 clients concurrently and crash the primary after the system runs for 10 seconds in a stable state. The timeout set for the system to elect a new primary is 10 seconds. After the new primary takes over and the system is again in a stable state, we recover the old primary. The old primary will catch up by synchronizing the missing oplog entries.

Figure 7 shows the throughput of the system during failover when chaining is enabled. The new primary steps up after losing the old primary over an election timeout. The clients can discover the new primary and issue writes to it immediately, thus the throughput resumes to the level before the failover. When the old primary is recovering (at 40s), it fetches the missing oplog entries from another secondary in this case. In our tests not shown here with chaining disabled, the old primary catching up its oplog from the new primary increases the workload on the new primary but doesn't affect the performance since the new primary isn't saturated.

5.1.4 Comparing with Previous Implementation

We compare the latest released version 4.4 with a previous MongoDB version 3.6 released in 2017. Version 3.6 is the last version that supports the old and deprecated replication protocol, which works in most cases but is unproven. Additionally, it has severe limitations due to its strong assumptions about the deployment environment: all messages must be replied within 30 seconds or otherwise the nodes must have failed. Thus, it does not tolerate faults like network partitions and could suffer from a "split-brain" if such faults happen.

The main advantage of our current protocol is fault tolerance as it makes fewer assumptions of the deployment. In most cases, we observed comparable performances, as shown in Figure 8. The performance of the newer version is better when the system isn't saturated. We believe this is not only because of the algorithmic and engineering improvements of the replication system, but also thanks to optimizations of many other parts of the server, e.g., journaling.

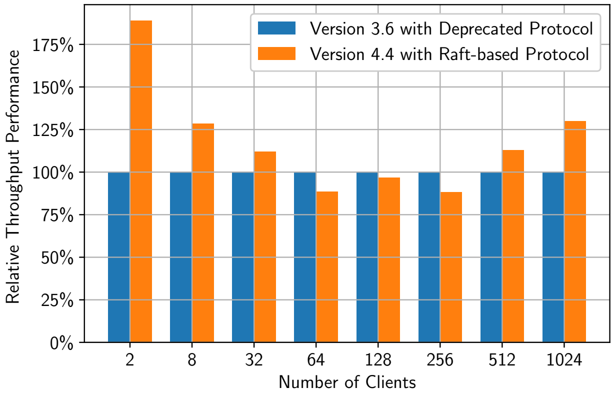

Figure 8: Relative throughput comparison between deprecated and Raft-base protocols

5.2 Metrics on MongoDB Atlas

To examine the performance of failover and its impact in production, we analyzed the failover metrics of MongoDB instances deployed on MongoDB Atlas, a hosted database as a service. When the data was collected in June 2020, almost all replica set instances were on the latest major releases: 3.6.18 (26.86%), 4.0.18 (44.17%) and 4.2.6 (28.54%). The vast majority of failovers (89.03%) are caused by planned maintenance, 6.15% by priority takeover (§4.5.2) and 4.82% by election timeout. We focused on those caused by planned maintenance and election timeout to measure the impact of expected and unexpected failovers.

5.2.1 Planned Maintenance

As part of rolling upgrade for planned maintenance on MongoDB Atlas, the old primary will be stepped down via a command. The stepdown command pauses new writes, waits for any eligible secondary to catch up and then asks the eligible secondary to step up, as discussed in §4.5.1. We measure the time duration from starting the election to when the new primary is available for new writes, referred to as *Local Write Unavailability*, and from starting election to when the first no-op write on stepup gets committed, referred to as *Majority Write Unavailability*. Note that the entire unavailable windows perceived by clients are longer since they start from when the old primary pauses writes. The extra unavailability window beyond our measurements heavily depends on the workload and is less comparable across replica sets.

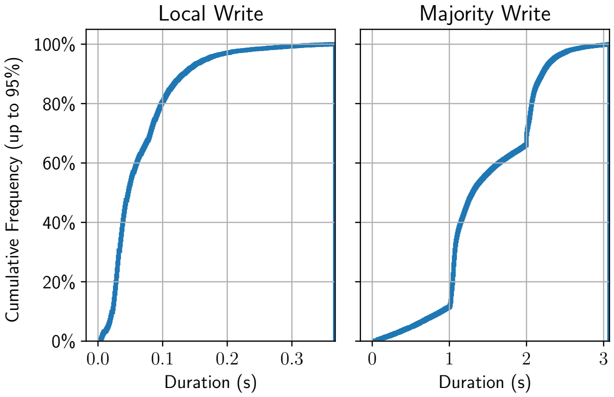

Figure 9: Unavailability due to planned maintenance

Figure 9 shows the cumulative frequency of local and majority write unavailability up to 95th-percentile, 0.37 seconds and 3.08 seconds respectively. Since an election only involves a few round-trips in the same data center, a candidate can finish its election quickly and start to accept new writes. However, it takes longer for other nodes to learn there is a new primary and start to sync from it directly or indirectly. We can observe two clusters of duration for majority writes: around 1 and 2 seconds. We believe this is due to the implementation of sync source selection based on heartbeats. We are actively working on delivering performance improvements in this area to all supported versions.

5.2.2 Election Timeout

When a secondary cannot see a primary for a given election timeout, it runs for election. Failovers caused by election timeout are the fault-tolerant scenarios for which the replication system is designed. We measure the same local and majority write unavailability as above. However, both measurements include the primary catchup phase (§4.3) which involves more work in election timeout cases, so both measurements are longer than that of planned maintenance. By contrast, it is essentially a no-op in planned maintenance since the old primary has waited already. Additionally, overloaded systems are a common reason of failover, which leads to longer primary catchup phases.

The perceived unavailable windows by clients are also longer since they start from when the old primary becomes unresponsive. Usually, they add about an election timeout (10 or 5 seconds on Atlas) to what we measured.

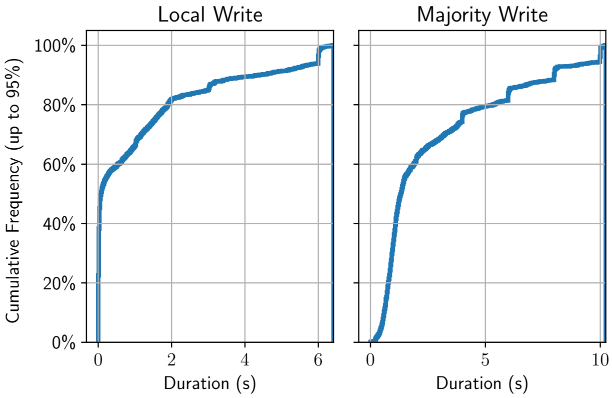

Figure 10: Unavailability due to election timeout

As shown in Figure 10, 95% of failovers due to election timeout start to accept writes within 6.41 seconds after election, and commit their first majority writes within 10.24 seconds. The sharp steps are also aligned with the default 2 second heartbeat intervals.

5.2.3 Replication Network Traffic

MongoDB's consensus protocol allows flexible replication paths and can save cross-datacenter traffic by allowing secondaries syncing from others from the same datacenter. We examined the replication network traffic on Atlas to estimate the cost of cross-datacenter traffic.

Figure 11 shows the distribution of daily replication network traffic on secondaries on Atlas during a week. While some replica sets have no writes and may mainly be used for reads, there are some others that generate gigabytes or terabytes of data. 50% of replica sets generate less than 8.08MB per day, and 95% generate less than 7.94GB per day. Figure 12 shows the distribution of the same data weighted by the daily replication network traffic. It is obvious that a small portion of replica sets generate disproportionately large amount of replication traffic. In fact, the top 5% of all replica sets account for more than 92.6% of all replication network traffic. Given the high traffic volume, it is valuable for a multi-node cross-datacenter deployment to minimize the cost of cross-datacenter traffic from the primary. The cost could be expensive (e.g., \$0.01~\$0.147 per GB on AWS [1]). MongoDB made it possible to minimize cross-datacenter traffic by allowing syncing from another node in the same datacenter instead of the primary.

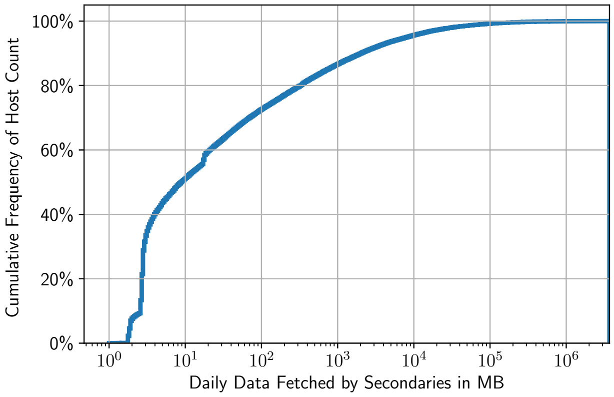

Figure 11: Distribution of daily replication network traffic

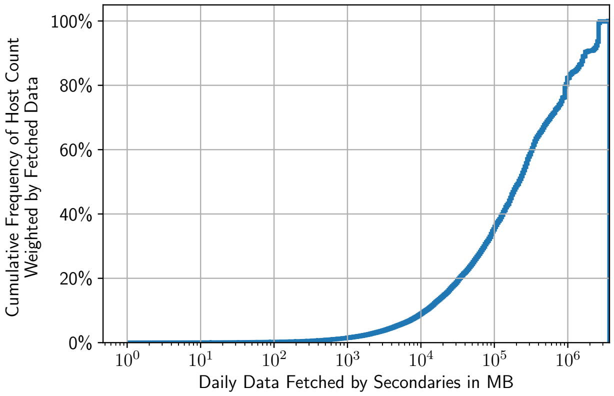

Figure 12: Weighted distribution of daily replication network traffic

6 Related Work

Replication and consensus are both very well-studied areas. This section reviews related works in two main categories: linearizable replication in production databases, and developments in consensus algorithms.

Linearizable replication in production databases. Many production databases build their replication systems via consensus protocols, mostly using Paxos or Raft. One recent popular Paxos-based system is Google's Spanner [7], and examples of Raft-based systems include TiDB [32], RethinkDB [28], CockroachDB [6, 31]. Another approach with similar data replication paths is the primary-backup scheme with external membership management (e.g., a consensus service). Recent database systems of this type include Aurora [35] and FaRM [8, 30]. To the best of our knowledge, these systems do not support pull-based data transfer in their data replication paths.

Another important scheme is Chain Replication (CR) [27]. It can achieve a chaining topology similar to MongoDB. The differences between CR and our proposed scheme are threefold. First, our scheme can support many types of topology and chaining is one typical use case. Second, in our scheme a request can commit after replicating to a majority, while in CR a request needs to replicate to all nodes before it commits. Third, similar to primary-backup schemes, CR needs a third-party to perform a safe leader (head) change. Nevertheless, CR can perform consistent read requests on the tail node, which can improve the system performance, while common consensus-based systems cannot.

CORFU [3] and Delos [2] have the clients (i.e., the learner role in Paxos) pull logs from the server; our approach takes one step further and has the servers (i.e., the acceptor role) pull logs from each other. During this process, we found that at least for Raft, modifying the protocols to enable servers to pull logs from other servers is challenging.

Study in consensus algorithms. Except for searching for more understandable basic consensus protocols, the developments in consensus algorithms can be classified into two categories: 1) optimizations that only require a minimal change to an existing protocol (usually Paxos); 2) new protocols built from scratch or heavily modifying a previous protocol.

The first category of works, in our experience, is closer to our needs in practice. PigPaxos [5] has a group of secondaries relay the messages from the primary to alleviate the primary bottleneck, which achieves similar flexibility as we do. Reconfiguration [18] of a replication group is a major challenge in MongoDB and we developed a refined version of reconfiguration in the latest release 4.4. Leases [10, 22] provide a way to allow consistent local reads in an efficient way. We are investigating this direction for future improvements. One thing we need to point out is that incorporating these or other optimizations of this category into MongoDB may be more difficult than into other systems that use stock Paxos protocols, because our protocol is a heavily modified version of Raft. However, it has been demonstrated [36] that such porting of optimizations can be achieved by drawing a one-to-one correspondence between each step of these protocols, e.g., by using refinement mapping.

It is an interesting and open question how MongoDB can benefit from the optimizations in the second category of works. These optimizations are often disruptive but effective. For example, to improve the performance for geo-replication, one can shard the log space [20], use fast quorums [23], or rethink the layering between replication and the rest of the system [15, 19, 37]. Replaying these works in MongoDB may require more efforts because it could suggest a complete reconstruction of the system.

7 Conclusion

In this paper we presented the design and implementation of the fault-tolerant and linearizable replication system in MongoDB. We proposed a novel pull-based consensus protocol that is a modification of Raft. With this pull-based scheme, MongoDB allows a more flexible control of data transmission paths. We described how this consensus protocol works, how MongoDB integrates it with the rest of the replication system, and the extensions of the replication protocol that support our rich feature set. We reported our evaluation on EC2 and our data analysis on MongoDB's cloud platform, and concluded that MongoDB can replicate data efficiently and reliably.

Acknowledgments

We thank our shepherd Mahesh Balakrishnan and anonymous reviewers at NSDI for providing valuable feedback. We thank Robert Guo for his guidance and assistance on performance testing. We thank William Schultz for contributing to the formal definition of majority commitment. We thank Bruce Lucas and Eric Milkie for helping investigate performance bottlenecks in the experiments. We thank the Data Engineering team and the Cloud team for collecting and maintaining the Atlas metrics and enabling the analysis of Atlas metrics. We thank our internal reviewers: Tess Avitabile, William Schultz, Andy Schwerin, Michael Cahill, Judah Schvimer, Robert Guo, and Asya Kamsky. Finally, we would like to thank the entire MongoDB Replication team and its alumni, all of whom contributed to the design and implementation of the protocol and features described in this paper.

References

[1] *Amazon EC2 On-Demand Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/.

[2] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, ET AL. Virtual Consensus in Delos. In *Proc. OSDI*. Oct. 2020.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, AND J. D. Davis CORFU: A Shared Log Design for Flash Clusters. In *Proc. NSDI*. Apr. 2012.

[4] M. Burrows The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. OSDI*. Nov. 2006.

[5] A. Charapko, A. Ailijiang, AND M. Demirbas PigPaxos: Devouring the communication bottlenecks in distributed consensus. *arXiv preprint arXiv:2003.07760* (2020).

[6] *CockroachDB*. http://www.cockroachlabs.com/.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, ET AL. Spanner: Google's globally distributed database. In *Proc. OSDI*. Oct. 2012.

[8] A. Dragojević, D. Narayanan, M. Castro, AND O. Hodson FaRM: Fast remote memory. In *Proc. NSDI*. Apr. 2014.

[9] M. J. Fischer, N. A. Lynch, AND M. S. Paterson Impossibility of distributed consensus with one faulty process. *JACM* 32, 2 (Apr. 1985).

[10] C. Gray, AND D. Cheriton Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (Nov. 1989).

[11] R. Guo Mongodb's javascript fuzzer. *ACM Queue* 15, 1 (Mar. 2017).

[12] M. P. Herlihy, AND J. M. Wing Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990).

[13] P. Hunt, M. Konar, F. P. Junqueira, AND B. Reed ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. ATC*. June 2010.

[14] A. Kamsky Adapting tpc-c benchmark to measure performance of multi-document transactions in MongoDB. *PVLDB* 12, 12 (2019).

[15] T. Kraska, G. Pang, M. J. Franklin, S. Madden, AND A. Fekete MDCC: multi-data center consistency. In *Proc. EuroSys*. Apr. 2013.

[16] F. Lai, M. Chowdhury, AND H. Madhyastha To relay or not to relay for inter-cloud transfers? In *HotCloud 18*. 2018.

[17] L. Lamport Paxos made simple. *SIGACT* 32, 4 (2001).

[18] L. Lamport, D. Malkhi, AND L. Zhou Reconfiguring a state machine. *SIGACT* 41, 1 (2010), 63–73.

[19] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, AND A. El Abbadi Low-latency multi-datacenter databases using replicated commit. *PVLDB* 6, 9 (July 2013).

[20] Y. Mao, F. P. Junqueira, AND K. Marzullo Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI*. Dec. 2008.

[21] *MongoDB Atlas*. https://www.mongodb.com/cloud/atlas.

[22] I. Moraru, D. G. Andersen, AND M. Kaminsky Paxos quorum leases: Fast reads without sacrificing writes. In *Proc. SoCC*. Nov. 2014.

[23] I. Moraru, D. G. Andersen, AND M. Kaminsky There is more consensus in egalitarian parliaments. In *Proc. SOSP*. Nov. 2013.

[24] B. M. Oki, AND B. H. Liskov Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC*. June 1988.

[25] D. Ongaro Consensus: Bridging theory and practice. PhD thesis. Stanford University, 2014.

[26] D. Ongaro, AND J. K. Ousterhout In search of an understandable consensus algorithm. In *Proc. ATC*. June 2014.

[27] R. van Renesse, AND F. B. Schneider Chain replication for supporting high throughput and availability. In *Proc. OSDI*. Dec. 2004.

[28] *RethinkDB*. http://www.rethinkdb.com/.

[29] W. Schultz, T. Avitabile, AND A. Cabral Tunable consistency in MongoDB. *PVLDB* 12, 12 (Aug. 2019).

[30] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, AND M. Castro Fast general distributed transactions with opacity. In *Proc. SIGMOD*. June 2019.

[31] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, ET AL. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proc. SIGMOD*. June 2020.

[32] *TiDB*. http://www.pingcap.com/.

[33] *TLA+ Specification of MongoDB Replication Protocol*. https://github.com/mongodb/mongo/blob/master/src/mongo/db/repl/tla_plus/RaftMongo/RaftMongo.tla.

[34] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, AND J. Mulrow Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *Proc. SIGMOD*. June 2019.

[35] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, AND X. Bao Amazon Aurora : Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proc. SIGMOD*. May 2017.

[36] Z. Wang, C. Zhao, S. Mu, H. Chen, AND J. Li On the parallels between Paxos and Raft, and how to port optimizations. In *Proc. PODC*. July 2019.

[37] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, AND D. R. K. Ports Building consistent transactions with inconsistent replication. In *Proc. SOSP*. Oct. 2015.

Appendix A: Comparison between Raft and MongoDB Consensus

Raft Protocol

Minor changes are marked in *italics*, e.g., changing from index to timestamp, comparison of OpTimes (<term, timestamp> pairs) and introducing heartbeats; major behavioral changes are marked in red.

State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)

votedFor candidateId that received vote in current term (or null if none)

log[] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)

lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)

matchIndex[] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

(Invoked by leader to replicate log entries; also used as heartbeat.)

Arguments:

term leader's term

prevLogIndex index of log entry immediately preceding new ones

prevLogTerm term of prevLogIndex entry

entries[] log entries to store (empty for heartbeat; may send more than one for efficiency)

leaderCommit leader's commitIndex

Results:

term currentTerm, for leader to update itself

success true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term > currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit < commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

term candidate's term

candidateId candidate requesting vote

lastLogIndex index of candidate's last log entry

lastLogTerm term of candidate's last log entry

Results:

term currentTerm, for candidate to update itself

voteGranted true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote

Rules for Servers

All Servers

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

Followers

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders

- *Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts*
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ⩾ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ⩾ N, and log[N].term == currentTerm: set commitIndex = N

MongoDB Consensus Protocol

State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)

votedFor candidateId that received vote in current term (or null if none)

log[] log entries; each entry contains command for state machine, *timestamp* and term when entry was received by leader

Volatile state on all servers:

lastCommitted *OpTime* of highest log entry known to be committed (initialized to minimum, increases monotonically)

lastApplied *OpTime* of highest log entry applied to state machine (*initialized to last log entry's OpTime*, increases monotonically)

lastPosition[] for each server, OpTime of highest log entry known to be replicated on that server

RequestVote RPC (Omitted)

. . . The same as Raft's RequestVote RPC *except changing index to timestamp* . . .

PullEntries RPC

(Replicate log entries from its sync source.)

Arguments:

prevLogTimestamp
timestamp of last fetched log entry.

Results:

entries[] log entries with a timestamp greater than or equal to prevLogTimestamp

commitPoint sync source's lastCommitted

Receiver implementation:

1. Return the log entries with a timestamp greater than or equal to prevLogTimestamp. Could be empty if no such entry exists.

Sender implementation after RPC call:

1. If returned entries is empty or last OpTime in entries is less than last OpTime in log, select a new sync source and retry PullEntries.
2. If last log entry conflicts with the first of entries (*due to different OpTimes*)
   (a) Traverse the log on sync source backwards until a common entry is found
   (b) Delete all existing entries following the common entry
   (c) Roll back the data to the state right after the common entry
3. Append any new entries not already in the log
4. If commitPoint > lastCommitted, set lastCommitted = min(commitPoint, *OpTime* of last new entry)

UpdatePosition RPC

(Sending latest positions of all known nodes to sync source.)

Arguments:

term	currentTerm, for leader to update itself
position[]	lastPosition[] on the sender

Results: None

Receiver implementation:

1. Merge position and lastPosition to record the highest known position for each member
2. Send UpdatePosition RPC to sync source if the receiver has one

Heartbeat RPC

(Used for liveness monitoring, commit point propagation, and sync source selection.)

Arguments:

term	sender's term
senderId	sender's node Id
role	sender's role
position	sender's last log entry's OpTime
commitPoint	sender's lastCommitted

Receiver implementation

1. Record the role and the current time of the last heartbeat for senderId for liveness monitoring
2. Update lastPositions[senderId] to position if position is higher, for sync source selection
3. If commitPoint > lastCommitted, set lastCommitted = min(commitPoint, *OpTime* of last new entry)

Rules for Servers

All Servers

- Apply log entries speculatively when appending them to log
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

Followers

- Respond to RPCs from candidates and leaders
- If election timeout elapses without *receiving Heartbeat RPC from current leader*: convert to candidate

Candidates

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If election timeout elapses: start new election

Leaders

- If command received from client: append entry to local log, respond after lastCommitted > entry's OpTime.
- *If there exists an entry such that entry.OpTime > lastCommitted, a majority of lastPosition[i] ⩾ entry.OpTime, and entry.term == currentTerm: set lastCommitted = entry.OpTime*

Appendix B: TPC-C Experiments

Setup. This section uses 3-way replication with 3 server VMs and 1 client VM. All VMs are deployed in the same Availability Zone. Each client thread continuously inserts documents of 1000 bytes in a closed-loop. We vary the number of client threads to control the offered load in the system.

	chaning disabled			chaining enabled		
	throughput (ops/s)	50% latency (ms)	95% latency (ms)	throughput (ops/s)	50% latency (ms)	95% latency (ms)
DELIVERY	58.16	144.14	257.72	60.21	129.94	247.22
NEW_ORDER	650.33	70.68	153.64	675.33	66.2	141.09
ORDER_STATUS	58.01	30.32	56.17	59.71	32.95	62.54
PAYMENT	628.51	37.04	169.28	651.46	37.91	171.52
STOCK_LEVEL	58.46	7.25	22.6	60.28	7.49	23.88
TOTAL	1453.47			1506.98		

Table 1: Results of an adapted TPCC benchmark. The benchmark denormalizes the data and leverages MongoDB query language and transaction semantics to be consistent with MongoDB best practices [14]. The test is run with 100 client threads and 100 warehouses on the same replica set setting as above without any bandwidth limit. Since the TPCC workload is CPU-bound, the performances of both chaining enabled and disabled settings are very close. Chaining-enabled case performs slightly better because it offloaded one secondary's oplog reading from the primary to a secondary and saved the CPU on the primary. The network was not saturated by replication: the chaining-disabled primary sent 15.56 MB/s in total over the network, including serving client requests and 4.05 MB/s to each secondary for oplog replication, while the chaining-enabled primary sent 11.70 MB/s in total. The gap of primary's network traffic is aligned well with the saved oplog replication traffic.