MongoDB 3.4.0-rc3

2017-02-07

In April 2015, we discussed stale and dirty reads in MongoDB 2.6.7. However, writes appeared to be safe; update-only workloads with majority write concern were linearizable. This conclusion was not entirely correct. In this Jepsen analysis, we develop new tests which show the MongoDB v0 replication protocol is intrinsically unsafe, allowing the loss of majority-committed documents. In addition, we show that the new v1 replication protocol has multiple bugs, allowing data loss in all versions up to MongoDB 3.2.11 and 3.4.0-rc4. While the v0 protocol remains broken, patches for v1 are available in MongoDB 3.2.12 and 3.4.0, and now pass the expanded Jepsen test suite. This work was funded by MongoDB, and conducted in accordance with the Jepsen ethics policy.

1 Background

In the past year and a half, MongoDB has put a good deal of work into improved read safety, enabled by the adoption of their new replication protocol and the WiredTiger storage engine. Dirty reads were addressed in 3.2 by introducing a majority read concern, and stale reads were addressed with the introduction of the linearizable read concern in 3.4. MongoDB contracted with Jepsen to analyze the safety of these mechanisms, and adopted Jepsen linearizability tests as a part of their continuous integration suite.

In November 2016, MongoDB requested Jepsen perform a followup analysis of a 3.4.0 release candidate (3.4.0-rc3) to confirm whether their linearizable read concern behaved as designed. We found critical design flaws in MongoDB's old replication protocol (v0), and multiple bugs in the new replication protocol (v1). These errors allowed (and, for any cluster running the v0 protocol, still allow) MongoDB to lose acknowledged updates even at the strongest (majority) level of write safety. We'll begin with the v0 replication protocol, show how its successor v1 addresses its design flaws, then discuss the bugs we found in v1—and their respective resolutions in 3.2.12 and 3.4.0.

2 Protocol Version 0

MongoDB's original replication protocol aims to provide consensus over a log of database operations: the *oplog*. In each replica set, nodes vote to elect a *primary* node, which will lead changes to the oplog. That primary accepts and orders client operations (e.g. insert a document, increment some field in a given document, etc.), appends them to its oplog, and applies those operations in order. *Secondary* nodes periodically request fragments of the oplog from the primary[1], saying "I have oplog entries through time `t`; please provide all subsequent operations." This ensures that secondaries eventually receive the same log entries as the primary.

If two nodes have the same oplog, and operations are deterministic, then both nodes can independently apply log operations to their local state to obtain identical successor states. Consensus on operations in the log therefore provides a *replicated state machine*. In MongoDB, that state machine is the storage engine used for documents and indices—usually an mmap'ed file or WiredTiger.

This algorithm works so long as nodes never fail. However, our primaries likely *will* fail, and we will need to elect new ones. With multiple primaries, our job becomes harder: how do we ensure that oplog entries are *stable*? Imagine a secondary with oplog entries `[a b c]` discovers a primary with oplog `[a b d e]`. That secondary cannot blithely continue appending entries, because the two replicas have *diverged*: `c` vs `d`. Nodes could return different responses to queries—and subsequent operations, like `e`, might not apply cleanly. We must instead *roll back* the state machine to the last common point `[a b]`, and then apply the new primary's operations `[d e]`. In fact, MongoDB does exactly

[1] Secondaries in MongoDB are allowed to pull oplogs from other secondaries, not just the primary; so long as the secondary's oplog is more up to date. We assume, for clarity, that operations are directly replicated from the primary.

this: when a secondary detects its oplog has diverged, it identifies affected documents, dumps their current state to a file on disk, catches up to the primary, then replaces those documents with copies taken from the current primary. Then it can proceed as normal.

This creates some difficulties. In particular, since nodes apply operations to their state machines immediately, our secondary will have applied `c` already, and could have responded to queries with `c`'s effects. If `c` inserted a document, that document would be visible to clients, then *lost*. Clients could interpret this as a lost update or a dirty read, depending on how the insertion of `c` completed.

For this reason, distributed logs typically maintain a *commit point*: the furthest index in the log which is known to be stable. Operations *before* the commit point will never be undone. Operations *after* the commit point are in flux: they may or may not become durable. If we can provide this property, we can avoid the rollback problem by only applying committed operations to the state machine. MongoDB *doesn't* do this—it applies operations as soon as they're received. This is why you can't trust the state on any given replica: dirty reads are a consequence of applying non-durable operations.

Protocol v0 can't prevent dirty reads, but it *can* prevent lost updates by deferring a successful response until it can *prove* that the given request will be durable. If an update doesn't succeed, we can't blame the database for losing it. If we perform updates with the *majority* write concern, MongoDB will not acknowledge our update until a majority of nodes have accepted it. The question becomes: is an operation replicated to a majority of nodes durable? Equivalently, is it possible for any node to become a primary *without* that operation?

Clearly, durability hinges on how we vote for primaries. We cannot elect any node at random, because we might choose a node from the minority which did *not* receive that operation. We must pick, informally speaking, *the newest node* available. But which node is newest?

In VoltDB 6.3, the node with the longest log was presumed to be authoritative. However, this is no guarantee of correctness: we could perform many unacknowledged operations on an isolated primary, while a fully connected one performs fewer, majority-acknowledged, operations. Picking the longer log would throw away those successful operations and preserve the failed ones!

Instead, MongoDB assigns a number to each log entry: the *optime*. Nodes will veto the election of any candidate with a lower optime than them, which implies leaders have an optime *at least as high* as any majority-committed operation. Optimes are monotonic on any given node: each operation the primary appends to the oplog has a strictly higher optime than the previous one. However, monotonicity is not sufficient. For instance, we could assign sequential integers as optimes, which has exactly the same problem as the longest-log approach: isolated nodes with more attempted ops could win elections.

To address this, MongoDB also pins optimes to the local system clock (incremented to preserve monotonicity where necessary). If we assume that clocks are perfectly synchronized, this implies that a new leader will have all operations from the most recent primary to make updates. There is, however, a problem with this approach: just as with the longest-log strategy, we have no guarantee that the *chronologically* most recent primary is actually *authoritative*.

Let nodes A, B, and C form a replica set, and let A be a primary node, which has become isolated from the cluster for 10 seconds. At wall-clock time 10, A receives two writes *w1* and *w2* from clients, which it cannot replicate, but nonetheless appends to its oplog at optimes `10` and `11` respectively. Concurrently, node B wins an election without *w1* and *w2*, and processes a single write *w3*, which it assigns optime `10`, and replicates to node C. Now isolate B, and rejoin A. If A and C hold an election, A will have the higher optime (`11` vs `10`) and will become the new primary. Failed writes *w1* and *w2* are preserved, and the majority-acknowledged write *w3* is lost.

This scenario may seem unlikely, requiring carefully selected partitions to occur in quick succession, but recall that in real computers, our clocks are *not* well-synchronized. They may differ by seconds, weeks, or even years. This widens the window of concurrency for data loss whenever primaries are isolated. The broader the clock drift, the more likely we are to lose data by electing an outdated node. The window is somewhat limited by the fact that secondaries will advance their clock to the primary's clock once they receive operations from it, but given time to run independently, an isolated primary with a fast clock will eventually outrace a majority-connected primary with a slower clock.

In short, MongoDB's v0 replication protocol does not provide consensus: it relies on synchronized clocks, and even where clocks are synchronized, can become skewed when many operations arrive at the same timestamp.

3 Verification

As before, we aim to verify that MongoDB provides linearizability on individual documents by performing a mix of reads, writes, and compare-and-set operations across a five node cluster, which comprises a single, non-sharded replica set.

Previous Jepsen tests of MongoDB waited a good deal of time between successive faults, and allowed perfectly synchronized clocks. While they exposed other errors around dirty and stale reads, they failed to identify the potential for lost updates described above. To reliably expose that behavior, we'll redesign Jepsen's nemesis, which introduces faults into the cluster—to specifically create isolated primaries with a fast clock.

We proceed in three phases: *isolate*, *kill*, and *stop*. To isolate, we identify all nodes which consider themselves a primary, and cut off their connections to the rest of the cluster. Then we advance those nodes' clocks by two minutes, so that any requests they receive will have an optime significantly higher than the rest of the cluster. Depending on priorities and timeouts, MongoDB may opt to preserve a current primary instead of undergoing a fresh election. To force an election every time, we kill all primaries before recovering. Finally, we recover the replica set by resetting all clocks, stopping all network partitions, and restarting all downed nodes.

We'll run this nemesis by repeatedly isolating all primaries, waiting 30 seconds for their state to diverge, then killing primaries and immediately recovering the cluster, followed by another 30 seconds for recovery.

Jepsen's linearizability tests are sufficient to show this fault, but because their verification is expensive, they perform only a few writes per second. We can find errors faster by designing a test specifically for lost updates. We repeatedly insert documents using majority write concern, while the nemesis isolates, kills, and recovers the cluster. After a quiescence period, we perform a few final read operations to check and see what documents are still present. If any successfully inserted documents are missing from the final read, we can conclude that MongoDB lost updates.

Protocol version 0 fails this test spectacularly: out of 4525 attempted inserts, 322 were preserved, and 93 were acknowledged, then lost.

```
{:valid? false,
 :lost
 "#{2492 2495 2502 2504 2509 3293 3300 3303 3312 3314 3317 3319 3326
    3328..3329 3335 3339..3340 3346 3348 3351..3353 3357 3362..3363 3367
    3370..3372 3376 3381..3382 3384 3388 3395 3400..3401 3405..3406 3408 3412
    3418 3424 3429..3431 3434 3438 3440 3443..3445 3447 3449..3450 3454 3456
    3459 3463 3468 3470 3474 3476..3478 3484 3492..3493 3495..3496 3499 3501
    3508 3510 3513..3514 3516 3521..3524 3526 3530 3536 3542 3545 3547 3549
    3552 3555..3556 3558}",
 :recovered
 "#{9 12..13 15 21 27 1353 1357 1360..1361 1366 1368 2078 2081 2084 2086
    2088..2089 3084..3085 3092 3098..3099 3103 4181 4184 4188..4189 4191
    4204}",
 :ok
 "#{...}",
 :recovered-frac  6/905,
 :unexpected-frac 0,
 :unexpected      "#{}",
 :lost-frac       93/4525,
 :ok-frac         322/4525}
```

To reiterate, this is a design flaw of the v0 replication protocol itself, not a bug; there are no plans to fix it. Every version of MongoDB, including the recently released 3.4.1 & 3.5.1, exhibits this behavior, and future versions likely will as well. The solution is to switch to MongoDB's newer replication protocol: v1.

4 Protocol Version 1

Faced with the unreliability of clocks—especially in virtualized environments—MongoDB has devised a second-generation replication algorithm which aims to address the design flaws in v0, as well as reducing failover times and improving election safety. Protocol

v1 is available in version 3.2 and higher, and is the default for newly created replica sets.

Version 1 adopts many features from the Raft consensus algorithm. Election IDs, derived from the candidate's monotonic wall clock, were added to v0 to allow clients to determine which primary was newer. However, because candidate wall clocks may not be synchronized, clients might assume old primaries were in fact newer. In v1, election IDs are replaced by logical *terms*: instead of being derived from the candidate's wall clock, and incrementing only on successful election, they are now integers incremented on every election attempt. Logical terms means nodes in v1 can only vote once per term; in v0, nodes are only prevented from double-voting by a 30-second cooldown, which is not entirely reliable.

In addition, optimes are no longer simple monotonic timestamps in v1, but a tuple of `[term, timestamp]`, which lets MongoDB preserve operations from the *logically* most recent primary, instead of the primary with the highest wall clock. In Raft, log entries are identified by sequential integer indices: 0, 1, 2, and so on. MongoDB uses non-contiguous, but still monotonic, timestamps as log indices. Monotonicity is assured because the election process requires that only nodes with the highest term from a successful election, and all majority-replicated entries from that term, can become primaries; and newly elected primaries constrain their generated timestamps to be at least as high as the most recent timestamp in the oplog. Where Raft ensures log contiguity by referring to the previous index, MongoDB refers to the previous timestamp. We can therefore form a bijection between Raft indices and MongoDB timestamps—MongoDB's v1 oplog is, in essence, a sparse Raft log. There are some differences—MongoDB secondaries can pull data from each other, instead of a primary, but we'll elide those here.

Where Raft applies only *committed* operations to its state machine, MongoDB applies operations as soon as they are received. Therefore, any replica in v1 may contain invalid state: dirty reads are still allowed, and rollbacks still occur. However, WiredTiger supports *snapshots* of the database, and MongoDB uses this to maintain a snapshot of the last known committed state. Queries at the *majority* read concern read from this (possibly stale) snapshot, rather than the dirty, most current state. With *linearizable* read concern, a primary performs a read from its uncommitted current state, then blocks until that state is committed, preventing both stale and dirty reads.

This snapshot approach comes with tradeoffs:[2] MongoDB can provide dirty but less stale reads on secondary nodes without the latency overhead of a fully linearizable read. However, the snapshot approach still requires rolling back the state machine when primaries diverge. Bugs in the rollback process have led to the loss of *committed* writes but these problems are generally being ironed out.

Unfortunately, as you probably already know, computers.

```
{:valid?          false,
 :lost            "#{1555 1557 1561 ... 4880..4881 4884}",
 :recovered       "#{365 906 912 ... 3716 3724..3726}",
 :ok              "#{178 182 184 ... 6069..6071 6075 6077}",
 :recovered-frac  16/3039,
 :unexpected-frac 0,
 :unexpected      "#{}",
 :lost-frac       139/2026,
 :ok-frac         515/6078}
```

In this test of MongoDB 3.4.0-rc3, with the v1 replication protocol, MongoDB lost 417 successfully inserted documents—even with majority write concern. That's almost half of the 932 acknowledged inserts (of 6078 attempts).

In Raft, primaries check the current term with every message: if the term changes, they know that a newer leader has come to power. Secondaries also use the term to verify that they're accepting writes from the most recent primary, ignoring messages from older terms. However, in 3.4.0-rc3, MongoDB could acknowledge writes from prior terms: the primary only checked to see that it was still a primary, rather than verifying that the term hadn't changed. In addition, heartbeat messages received by a stale primary could

[2]Note that this is not the same tradeoff made by other consensus systems like Consul, etcd, and Zookeeper. Those systems only apply *committed* updates to the state machine, but allow users to choose between linearizable reads which go through consensus, and sequentially consistent (e.g. stale) reads against any replica. Unlike MongoDB, those systems do not allow dirty reads.

allow it to advance its commit point, acknowledging writes which were not in fact durable. Both of these issues were patched in 3.4.0-rc4.

Unfortunately, 3.4.0-rc4 *also* lost acknowledged writes, because secondaries didn't check to see whether they were replicating from a newer node. They ignored terms and simply compared timestamps. Both of these bugs essentially stem from the legacy v0 protocol: some code that was written under v0's assumptions that timestamps indicated the most authoritative log, wasn't updated to reason using v1's logical terms.

This bug was fixed in 3.2.12, 3.4.0, and the development release 3.5.1: all three now preserve acknowledged writes in Jepsen's test of this isolation & clock-skew scenario. Of course, Jepsen cannot guarantee correctness, and additional bugs may be lurking in the code. For example, a recently discovered race condition in the voting process could allow secondaries to forget about votes, potentially voting twice if a node restarts during an election.

5 Discussion

MongoDB's version 0 replication protocol is inherently unsafe. Even in the just-released 3.4.1 it allows the loss of majority-acknowledged documents when primaries diverge with skewed clocks. Although v1 has been the default for newly created replica sets since MongoDB 3.2, there remain many production deployments of the v0 protocol, and I recommend they switch to v1 as soon as possible.

Protocol version 0 remains popular with users of arbiters, especially for three-datacenter deployments where one datacenter serves only as a tiebreaker. Specifically, when two DCs are partitioned, but an arbiter can see both, v1 allows the arbiter to flip-flop between voting for primaries in both datacenters, where v0 suppresses that flapping behavior. In both protocol versions, in order to preserve write availability in both datacenters, users *cannot* choose majority write concern. This means that when inter-DC partitions resolve, successful writes from one datacenter can be thrown away.

I recommend that two-DC users avoid arbiters in favor of standard replicas in a 2/3 or 4/1 split, with a majority in a primary DC. This allows the use of majority write concern with single-DC latencies in the primary DC, handles the loss of the secondary DC transparently, and in the event the primary DC fails, administrators can fail over (possibly losing a window of acknowledged writes) by reconfiguring the secondary DC's replica set. Users with three datacenters have a choice between keeping a majority in a single DC (lowering latencies in the happy case), or spreading nodes evenly between DCs (preserving safety and liveness in the event any DC fails, at the cost of inter-DC latency). If you *must* use arbiters, stick with protocol v0 to prevent flapping, but be cognizant of the risks.

Protocol v0 can also preserve more writes made with single-node write concern than v1. Of course, writes to a minority of nodes are *not* guaranteed to be durable, but many users prefer lower latency to safety, and want to preserve as many incomplete writes as possible. MongoDB is working on improving the v1 protocol to recover safely-applicable writes from reachable nodes during primary elections, which should make the v1 upgrade more practical. Users of 3.4.0 and higher can increase `settings.catchUpTimeoutMillis` to give newly elected primaries more time to scavenge incomplete writes.

Now, for version 1: users of the v1 replication protocol in 3.4.0-rc4 (and prior versions) are also vulnerable to data loss, due to multiple implementation bugs where terms were ignored in favor of wall-clock timestamps. The general release of 3.4.0 fixes these bugs, and additional fixes are available in 3.4.1. Users should upgrade to 3.4.1 to avoid these risks.

Note that while the v1 replication protocol adopts *ideas* from the Raft consensus algorithm, v1 is *not* Raft. The use of rollbacks, for instance, introduces new complexities. MongoDB also allows secondaries to replicate from other, more up-to-date secondaries, instead of from the primary. Furthermore, many of the atomic state transitions in Raft–for instance, voting–are not serialized through the log and applied to the state machine once committed; rather, MongoDB uses mutexes and direct updates of the storage system to handle meta-operations. If that mutex scope is not enforced carefully concurrency errors can result. MongoDB continues to refine their implementation, and will likely streamline it once v0 is deprecated.

As always, MongoDB users have a choice of write concern: how many nodes the primary will wait for before acknowledging a write. Many production users continue to use one- or two-node write concerns, instead of majority. Users should use majority write concern for any data they must keep; otherwise they run the risk of losing updates in a rollback.

MongoDB has devoted significant resources to improved safety in the past two years, and much of that ground-work is paying off in 3.2 and 3.4. Dirty reads, which we covered in the last Jepsen post, can now be avoided by using the WiredTiger storage engine and

selecting `majority` read concern. Because dirty reads can be written back to the database in read-modify-update cycles, potentially causing the loss of committed writes, users of ODMs and other data mappers should take particular care to use majority reads unless making careful use of findAndModify.

Finally, stale reads are preventable in MongoDB 3.4.0 and higher through the use of the linearizable read concern. Users should use linearizable reads to ensure the visibility of writes when communicating via side-channels, or whenever the most recent *committed* state is required. When the most recent *uncommitted*, possibly invalid state will suffice, a standard read on a primary will have lower latency.

This table shows the consistency anomalies possible in recent releases of MongoDB when the strongest options are used (protocol v1, majority writes, linearizable reads). Majority reads were introduced in 3.2.x and should have prohibited dirty reads, but given committed writes could be lost in 3.2.11, it's unclear whether preventing dirty reads is all that meaningful. Linearizable reads were introduced in 3.4.x, but again, are not particularly meaningful when committed writes can be rolled back.

Version	Lost updates	Dirty Reads	Stale Reads
3.0.14	Allowed (no v1)	Allowed (no maj. read)	Allowed (no lin. read)
3.2.11	Allowed (v1 bugs)	Kinda	Allowed (no lin. read)
3.2.12	Prevented	Prevented	Allowed (no lin. read)
3.4.0-rc3	Allowed (v1 bugs)	Kinda	Kinda
3.4.0-rc4	Allowed (v1 bugs)	Kinda	Kinda
3.4.0	Prevented	Prevented	Prevented

The v0 protocol (and of course, any test with non-majority writes) continues to lose data in Jepsen tests. With the v1 protocol, majority writes, and linearizable reads, MongoDB 3.4.1 (and the current development release, 3.5.1) pass all MongoDB Jepsen tests: both preserving inserted documents, and linearizable single-document reads, writes, and conditional updates. These results hold during general network partitions, and the isolated & clock-skewed primary scenario outlined above. Future work could explore the impact of node crashes and restarts, and more general partition scenarios.

This research was funded by MongoDB, and conducted in accordance with the Jepsen ethics policy. My sincerest thanks to the MongoDB team for their assistance—especially Dan Pasette, Jonathan Abrahams, Eric Milkie, and Andy Schwerin.