Tokutek®

A New Consensus Algorithm for TokuMX and MongoDB

By Zardosht Kasheff
Software Developer, Tokutek
@zkasheff

A Technology White Paper
Fall 2014

Executive Summary

Ark is an implementation of a consensus algorithm similar to Paxos and Raft. It is designed to improve replica set failover in the Tokutek distribution of MongoDB: TokuMX™. While it has many similarities to Raft, it also has some important differences. Ark was designed by Tokutek to fix known issues with the election protocol used in basic MongoDB, and in the Tokutek distribution of MongoDB, TokuMX, as well. Today, the existing protocol is neither AP nor CP, when considered in the context of the CAP theorem. Tokutek will eliminate these inherited issues in TokuMX using Ark.

Tokutek set out primarily to modify the election protocol to make TokuMX a true CP system. That is, in the face of network partitions, TokuMX will maintain consistency. To do so means ensuring that any write that is successfully acknowledged with majority write concern is never lost.

The high level goal of Ark is to improve failover behavior. A Tokutek tech report that details the issues encountered and the approach Tokutek will use for fixing them, can be found at: http://arxiv.org/pdf/1407.4765v1.pdf.

The Ark "fix" will be included in TokuMX v2.0.

Explaining Ark: The Basics

Ark is a consensus algorithm similar to Raft and Paxos that Tokutek has developed for TokuMX. The purpose of Ark is to fix known issues in elections and failover. There are three important components to examine here:

- elections/failover
- replication rollback
- write acknowledgement

First, for those unfamiliar with MongoDB, let's start with the basics.

What is MongoDB's replication model?

The purpose of replication is to have multiple machines, or members, store the exact same data. It works as follows:

- One member in the set is designated as primary.
- The primary is the only member that accepts writes from users. Details of the write are written to the oplog (similar to the binary log in other databases, this can also be thought of as the replication log).
- All other members in the set are secondaries.
- Entries in the oplog have a position associated with them that defines the order of writes. Secondaries apply modifications in the order they appear in the oplog.
- Secondaries cannot be modified by users. Secondaries constantly pull data that originates from the primary's oplog, save the data in its own oplog, and apply it locally.

Notably, the secondary does not have to pull data directly from the primary. Any secondary is allowed to pull data from any other member in the replica set, as long as the other member's position is ahead of the secondary's position.

The group of members that make up the primary and secondaries is called a "replica set." Additionally, note that MongoDB replication is asynchronous. Secondaries are responsible for pulling data off other members that are further ahead of them, and announcing their progress. Primaries are not responsible for pushing data to secondaries to ensure that secondaries are up to date.

Elections and Failover

One design goal for MongoDB replication is to be highly available. Should the primary become unavailable due to a crash, getting disconnected from the network, or some other reason, the rest of the replica set ought to notice and work together to select a new primary, so that the system may resume accepting writes. The process by which they arrive at a consensus to select a new primary is called, naturally, a consensus algorithm.

If the primary becomes unreachable by other members of the replica set, the other members will try to hold an “election,” a process by which they choose a new primary to start accepting user writes. A majority of members in the replica set are required to elect a new primary. Note that majority means greater than half, so if there are four members in the replica set, three are needed to elect a primary.

Every two seconds, all members exchange information with each other, in what are called heartbeats. The two most important things exchanged are the member’s current oplog position and the fact that a member is still up and responding. In MongoDB, a replica set member (be it primary, secondary, or arbiter) is deemed unreachable if a heartbeat fails.

A typical election takes place as follows:

- The primary loses network connectivity and becomes disconnected from the replica set.
- A secondary sends out heartbeats and doesn’t receive a response from the primary.
- The secondary decides the primary is unavailable and works on getting itself elected.

Replication Rollback

Another important concept is rollback. Take the following scenario:

- A member P gets some write applied.
- Due to some network partition, a failover happens, and new primary is elected.
- At the time of failover, the write that was just applied to P had not been replicated to any secondary.
- At some point, when P gets reconnected to the set, P will “roll back,” or undo the write it just did, because that write is not part of the new replication stream.

Due to the design decision to make MongoDB replication asynchronous, rollback is always a possibility.

Write Acknowledgement

Traditionally, in single-node databases without replication, users consider data safe if a copy of the write operation is written to some recovery log (MongoDB calls this the journal), and the log is synced to disk. In a highly available system, this is ineffective because if the primary becomes unavailable and a failover happens, it’s irrelevant whether the data is in the primary’s recovery log since it still may be rolled back.

What is relevant is whether the data is on the machine that takes over during failover. The MongoDB concept of “write concern” gives the user an acknowledgement that a write has successfully been replicated to a member. So, if we have a replica set of five members, and a user requests a write acknowledgement from two members, or, “w : 2,” this means the user is waiting to know that his write has successfully made it to two members of the set. A write concern of “w : majority” means the user is waiting to know that his write has successfully made it to a majority of members.

While users have a spectrum of choices for write concern (Eliot Horowitz, CTO of MongoDB, has a good presentation on this topic), the write concern that we are most interested in is “majority.” Theoretically, if a write has been acknowledged by a majority of the replica set, then any elected member ought to include that write. Here is why:

- Any successful election will be run by a majority of the replica set.
- Elections ought to elect the member that has replicated the most data.
- If a write has been acknowledged by a majority of the replica set, then SOME member participating in the successful election has replicated and acknowledged the write. Therefore, anyone who gets elected ought to have replicated the write.

However, this is not currently the case. Ensuring this property is Ark’s biggest goal.

How Elections and Failover Work

To better understand how elections work, it’s instructive to look at the threading model. The threads that impact failover are:

- Data sync threads: These threads are responsible for copying changes from some other member in the replica set and acknowledging their arrival. This is the central purpose of replication.
- Heartbeat threads: Every two seconds, a member requests a heartbeat from another member to get status information, such as oplog position, whether they are a primary, and to verify that they are available. So, in a five-node replica set, each member has four threads, each requesting a heartbeat from another member in the set every two seconds.
- Manager thread: As far as failovers are concerned, this thread decides to run an election and attempts . to transition itself from secondary to primary.
- Voting threads: These threads handle requests from other members in the replica set that aim to elect themselves as primary. A member may be handling a vote request from another member on a voting thread while simultaneously trying to elect itself on the manager thread.

An important thing to note: these threads work mostly independently from each other. While the following events are unlikely, they are possible:

- Member A may be replicating off of member B even though A’s manager thread is acting as though B is down.
- Member A may be running an election even though it thinks the primary is up.
- Member A may be voting to replace a primary even while it is syncing from that primary.

The threading model is both a blessing and a curse. It's a blessing in that the responsibilities of the various threads are very simple to understand. For example, the data sync threads simply replicate data. That is it. They don't concern themselves with whether an election is taking place, so there's little complexity in their inner workings. The model is a curse in that there is very little synchronization or shared information between the threads. Elections, at a high level, depend on information in the oplog, such as "how far along is the oplog," and this may change during the election.

The next step to examine is the existing election protocol. The current primary member will be called OP (for original primary), and what will become the new primary (because it is ahead of everyone else) member NP (for new primary).

If there is a network partition such that OP is disconnected from the replica set, at a high level, two independent things must happen:

- NP notices that it cannot reach OP, looks at what it knows of the state of the replica set (via heartbeat messages it has received), and says "I think I'll make a good primary." NP then proceeds to elect itself.
- Independently, OP notices that it cannot reach a majority of the set, and decides to transition from primary → secondary.

What if one of these steps doesn't happen? Specifically, what happens if NP successfully elects itself because OP is temporarily disconnected, but OP never gets around transitioning to secondary before getting reconnected to the replica set? In short, we temporarily have two primaries.

How does NP goes about electing itself? The current election process has two phases. The Ark tech report refers to these as a speculative election and an authoritative election.

In the first phase, NP asks every other member in the set, "I believe the primary is down and that I ought to step up and become the new primary. If I proceed to try to elect myself, will you vote yes?" The purpose of this speculative election is to give NP a good idea of whether an actual (authoritative) election will succeed. In the second phase, NP asks every other member in the set, "I want to become primary, please vote," and if a majority of votes come back saying "yes," NP becomes primary.

To understand how these phases work in more detail, it's important to discuss the second phase, the authoritative election, first. In an authoritative election, NP has decided to try to become primary and sent messages to every other member requesting a vote. Other members process this vote on voting threads. When each member gets this message, it does the following:

- Each member has the choice of voting "yes," "no," or "veto." If any member votes "veto," the election fails, regardless of whether a majority voted "yes." Reasons a member may choose to veto (e.g. NP's config version is stale) are not interesting for this discussion.

- If the member participating in the election has voted “yes” for anyone in the last 30 seconds, then vote “no.” A member may vote “yes” only once every 30 seconds.
- Otherwise, vote “yes.” Don’t bother looking at the oplog’s position. This last fact is strange, but true, and is pointed out in https://groups.google.com/forum/?hl=en-US&fromgroups#!topic/mongodb-dev/lH3hs8h7NrE.
    - Note that by voting “yes” this member will not vote “yes” for anyone else in 30 seconds. This may be to make it difficult for two primaries to be elected in a short period of time.

If NP gets a majority of “yes” votes, NP becomes primary.

The authoritative election is discussed first to demonstrate the following point: while successful elections are great, failed elections may have severe consequences. A failed election may have a non-negligible (but non-majority) subset of the replica set vote yes, and as a result disqualify themselves from voting in other elections for 30 seconds. This makes having successful elections in the next 30 seconds more difficult, and can lead to long periods of downtime while the set struggles to elect a new primary.

Because failed authoritative elections may have severe consequences, they should only be run when there is good reason to believe that they will be successful. That is the purpose of the speculative election: to give NP good reason to believe its election will be successful, greatly reducing the probability of having a failed authoritative election.

Now let’s discuss some of the details of the first phase. NP sends a message to all other members asking “should I try to elect myself?” With the replies NP learns:

- Whether anyone would veto the election, thereby making NP’s election impossible. If so, NP does not proceed to the second phase. One possible reason is that while NP does not see OP, the original primary, other members do. If that is the case, these other members will tell NP not to bother with an election.
- Whether any member has an oplog that is ahead of NP’s oplog. If so, NP does not try to elect itself. After all, we want elections to elect the secondary that is furthest ahead. At the moment, this is the only synchronization done between elections and data sync threads.

That’s the election protocol, as it stands today. So what can go wrong? Theoretically, quite a bit.

High Level Issues with the Elections

Problem 1: Problems with the existing implementation.

Charity Majors of Parse / Facebook, in a talk at MongoDB World (not the keynote), said that at times elections may take 5-7 minutes to elect a primary. This doesn't sound like the intended design and sounds like a bug. While developing Ark, Tokutek found issues with the existing MongoDB and TokuMX implementation that could theoretically lead to this behavior.

A key property of the design is essentially "use speculative elections to avoid running failed authoritative elections, because a failed authoritative election may lead to nobody getting elected for at least 30 seconds." The following bugs filed, which are now fixed with Ark, may lead to failed authoritative elections that speculative elections could have avoided:

- https://jira.mongodb.org/browse/SERVER-14382
- https://jira.mongodb.org/browse/SERVER-14531

Problem 2: Too little synchronization with replication threads.
Currently, elections and data sync threads only communicate when a voting thread responding to a speculative election peeks at the oplog to determine if NP is far enough ahead. This is not sufficient. Minimally, there ought to be some communication in the authoritative election. Additionally, while elections are happening, this member's replication threads may be replicating and acknowledging data. This replicated and acknowledged data may be later rolled back because of the election this member is participating in.

Problem 3: Dealing with multiple primaries
The election protocol is designed to make the probability of having multiple primaries very low. Nevertheless, there may be periods where multiple primaries exist. These situations need to be resolved predictably, and currently, they are not.

These latter two problems are what causes writes that get acknowledged with majority write concern to possibly get rolled back.

Why Data May Be Lost on a Failover

The best way to demonstrate how data may be lost is with examples. For each of our examples, suppose things start out as follows. There is a five-node replica set: n1, n2, n3, n4, and n5, with the following setup:

- n1 is primary
- n2 and n3 are syncing off of n1
- n4 and n5 are syncing off of n3

Example 1

Suppose the following sequence of events happens:

- A network partition happens such that (n1, n2) gets disconnected from (n3, n4, n5).
- n3 notices that it cannot reach n1, and proceeds to elect itself.
- n4 and n5 vote for n3, and as a result, n3 becomes primary.
- n3 performs write A, which gets replicated and acknowledged by n4 and n5, thus having been acknowledged by a majority of the replica set.
- n1 reconnects to the replica set, but still as a primary. It has yet to step down.

Now there are two primaries, n1 and n3. One needs to step down. If n1 steps down, nothing bad happens, but if n3 steps down, a write that was acknowledged with majority write concern will be lost.

In MongoDB 2.4, this was quite possible. In dual primary scenarios, which primary stepped down was essentially arbitrary. SERVER-9765 addressed this issue. Now, in 2.6, a primary uses the timestamp of the last election to determine which primary should step down. In the example above, because n3 was elected at a later time than n1, n3 ought to remain primary and n1 ought to step down. Because members may participate in an election once every 30 seconds, the minimum amount of time between successful elections ought to be 30 seconds. That being said, it's unclear how clock skew between different members of the replica set impacts this algorithm.

Example 2

This example expands a bit on the first one:

- A network partition happens such that (n1, n2) gets disconnected from (n3, n4, n5).
- n3 notices that it cannot reach n1, and proceeds to elect itself.
- n4 and n5 vote for n3, and as a result, n3 becomes primary.
- n3 performs write A, which gets replicated and acknowledged by n4 and n5, thus having been acknowledged by a majority of the replica set.
- n1 reconnects to the replica set, but still as a primary. It has yet to step down.
- n1 accepts a write B that gets replicated and acknowledged by n2.
- n4 stops replicating off of n3 and starts replicating off n1.
- n4 rolls back write A, because n1 doesn't have it, and replicates and acknowledges write B.

Now there is a big problem. Both n1 and n3 are primary, and both have performed a different write that has been acknowledged by a majority of the replica set. One needs to step down and rollback their write. Whoever does will have rolled back a write that was acknowledged with majority write concern.

This example demonstrates an underlying problem of MongoDB's replication: a timestamp is used to determine oplog position (aka, GTID). The symptom this causes is that two different primaries may produce oplog entries whose positions interleave. Because write B happens after write A, as far as time goes, the timestamp stored in B's oplog entry on n1 is greater than A's oplog entry that has been replicated to n4. As a result, n4 thinks that n1 is farther ahead, and thinks it's ok to start replicating off n1. As soon as this decision is made, n4 will replicate and acknowledge B, which allows both writes to be acknowledged by a majority.

TokuMX does not use timestamps to determine oplog position in its current election algorithm (preceding Ark). As a result, the problem listed above does not impact TokuMX. However, TokuMX does have the problem where two different primaries produce oplog entries whose positions interleave. The existing 30-second timer between elections makes the problem harder to explain, but it does exist.

Nevertheless, solving these issues does not fix the election protocol. There is one more problem. Here is one final example.

Example 3

This example is similar to the preceding two, in that n1 is a primary, and after a partition, n3 becomes primary. If the resolution of dual primaries is robust and works, we can deterministically predict that the primary that was elected earlier will step down. Also, should the positions of oplog entries produced not intersect -- i.e., any entry produced by n3 (the later election) is guaranteed to come after any entry produced by n1 (the earlier election) -- then the following problem still exists.

- All secondaries replicate from n1.
- A network partition happens such that (n1, n2) gets disconnected from (n3, n4, n5).
- n3 notices that it cannot reach n1, and proceeds to elect itself.
- n4 and n5 vote for n3, but n3 has yet to become primary.
- After n4 and n5 have voted, the network restores itself.
- n1 performs a write A that is acknowledged by n2, n4, and n5 (4 out of 5 members!). n3 has yet to become primary or to replicate and acknowledge the write.
- n3 finally gets around to becoming primary, without having a chance to replicate and acknowledge A.
- n1 notices that n3 was elected later and steps down.
- all other members start replicating off n3, and as a result, rollback write A.

The problem here is that the process of write acknowledgement and voting for primaries are not communicating enough. n4 and n5 vote for n3, and then acknowledge a write that could be rolled back, in part because of the vote they just performed. This is the main underlying problem that TokuMX and MongoDB's current election protocol do not take into account, and this is the problem Ark rectifies.

So, in summary, the underlying issues in the existing election protocol are as follows:

First, when faced with dual primary situations, MongoDB and TokuMX must be able to resolve the situation predictably. This means:

- Two primaries never produce oplog entries whose positions interleave;
- When faced with two primaries, the one that produces oplog entries with smaller positions steps down.

Second, threads that sync data and acknowledge writes need to communicate more with threads that vote for primaries. In short, the following ought to happen:

- A member should not vote for a primary that will roll back a write it has acknowledged;
- A member should not acknowledge a write that may be rolled back due to an election it has already voted "yes" in.

Making sure these properties hold is what Ark does.

Fixing Majority Write Concern

Aside from miscellaneous behavioral improvements (such as SERVER-14382 or SERVER-14531), Ark's key behavioral changes are to fix the following two problems:

1. When a replica set has two primaries, the two primaries should never produce oplog entries whose positions interleave, and primary that produces smaller oplog positions should step down.

2. Threads that sync data and acknowledge writes need to communicate more with threads that vote for primaries:
   - A member should not vote for a primary that will roll back a write it has acknowledged;
   - A member should not acknowledge a write that may be rolled back due to an election it has already voted "yes" in.

Fixing Multiple Primaries

Primarily, Ark ensures that two primaries never produce oplog entries whose positions interleave. Here is how:

Unlike MongoDB, TokuMX does not use a timestamp as the oplog's position identifier, also known as a global transaction identifier or GTID. TokuMX uses a pair of sequence numbers. Let's call the GTIDs sequence numbers (*term*, *opid*). The GTID works as follows:

- Each oplog entry that a given primary produces will have a unique, increasing *opid*. So, member A may produce entries (5, 100), (5, 101), (5, 102)…

- Each successful election will have a unique *term*. When member A gets elected, it does so with the understanding that it will produce oplog entries with a *term* of 5, starting with (5,0).
- The GTID is compared lexicographically. For example, (5, 101) < (5, 102) < (6, 0).

If Ark can ensure that each successful election produces a unique *term*, two primaries never produce oplog entries whose positions interleave since GTID *terms* are compared first. Here is how Ark ensures that each successful election produces a unique *term*:

- Ark still has the two phases of elections referenced in part 2, the speculative and authoritative phase. In the authoritative phase, the member trying to vote itself primary does so with a proposed *term*. If member A has reached the authoritative phase and is trying to elect itself, it sends a message to all other members saying, "I wish to become primary with a *term* of 5."
- Each member voting in an election only votes yes for members with *terms* that are greater than any other *terms* it has voted for so far. So, if member B has already voted yes in an election for a member with a *term* of 13, it will automatically vote no in any subsequent election with a *term* <= 13, but is allowed to vote yes in any subsequent election with a *term* > 13.

These two simple rules ensure that no member votes yes in two separate elections with the same *term*. Because each successful election requires a majority, each pair of consecutive successful elections have at least one member in common. Therefore, no two successful elections will ever share the same *term*.

For those familiar with Raft, this *term* is similar to the election term described in Raft.
As a follow on to the description of how Ark ensures that two primaries never produce oplog entries whose positions interleave, here's how Ark ensures that the primary that produces smaller oplog entries steps down.

In MongoDB, a primary steps down under the following scenarios:

- It cannot reach a majority of the replica set.
- It sees another primary.

These conditions are not sufficient. As described in SERVER-9848, two primaries may see a majority of the set, but not each other. As a result, neither will step down. To fix this, Ark uses heartbeats to exchange election information between members. Thanks to heartbeats, members know the highest *term* that other members have voted for. If a primary sees that a member has voted for a higher *term* than its own, it steps down.

Now, dual primaries are guaranteed to resolve themselves properly. If two primaries exist, and both can reach a majority of the set, then both must have access to the highest *term* voted for in an election. This causes the primary with the smaller *term* to step down. If the primary with the smaller *term* doesn't see a higher *term*, then it must not see a majority of the set, and will step down for that reason. These changes cover how Ark resolves multiple primary situations.

Acknowledging Writes and Elections

The last problem is that of write acknowledgement. The current problem is not that secondaries accept writes that they should not, because writes can roll back. The problem is that secondaries blindly acknowledge any write they receive, regardless of whether the write may later be rolled back. To fix this, we do the following:

A member never votes for a primary that will roll back a write it has acknowledged:

- When requesting a vote, the primary shares its oplog position;
- If the primary's position is behind the voting member's current position it doesn't vote yes.

A member should not acknowledge a write that may be rolled back due to an election it has already voted "yes" in:

- Before acknowledging a write, a member checks to see if the write's *term* is less than the highest *term* for which it has already voted. If so, the write is not acknowledged.
- For example, suppose a member wants to acknowledge (5,100), because it has just processed that oplog entry. If it notices that it has already voted yes in an election with *term* 6, the write is not acknowledged. This is because the member cannot acknowledge if its yes vote will cause this write to roll back.

Conclusion

The Tokutek Ark algorithm ensures that writes that are acknowledged with a majority write concern never roll back.

Suppose a write has been acknowledged with majority write concern. Because each primary has its own *term*, we know that the only way this member loses its primary status is on some subsequent successful election with a higher *term*. That means the only way the write can roll back is because of a subsequent election. We know that any successful election must have a majority of "yes" votes, which must overlap with the majority that acknowledged this write. As established above, each member in the overlap will only vote yes if it knows the potential primary will not roll back the write, meaning the potential new primary must also have the write and will not roll it back.

The original MongoDB election algorithm would allow a member to acknowledge a write but also vote "yes" for a new primary that would roll it back. Therefore, those members in the overlap did not actually protect the write as their acknowledgement should have guaranteed they would.

About Zardosht Kasheff

Zardosht Kasheff has been a key member of the Tokutek development team since 2008. He was instrumental in the development of Fractal Tree® indexing for MongoDB. Prior to Tokutek, Zardosht worked as a software design engineer at Microsoft. He holds B.S. and M.S. degrees in Computer Science from the Massachusetts Institute of Technology. While at MIT, he pursued research on cache-oblivious dynamic search trees under Bradley Kuszmaul at the Computer Science and Artificial Intelligence Laboratory.

About Tokutek Inc.

Tokutek is a performance database company that delivers Big Data capabilities to leading open source data management platforms. Its breakthrough technology lets customers build a new class of applications that handle unprecedented amounts of incoming data and scale with the data processing needs of tomorrow. Tokutek applies patented Fractal Tree® indexing to increase MySQL performance and MongoDB performance, decrease database size, and minimize downtime. The company is headquartered in Lexington, MA, and has offices in New York, NY. For more information, visit tokutek.com or follow us on Twitter @Tokutek.