Dapr Blog

Text

MongoDB Replication: The Journal Journal, The Global Lock, & Durability

Posted on April 19, 2014 by Reuben Bond

MongoDB was designed from the ground up to be very simple to deploy and use. It's apparent, however, that less attention was paid to designing the architecture for a clustered deployment.

Clustered MongoDB hosts form replica sets for high availability. Within a replica set, there are primary replicas and secondary replicas. Primary replicas serve all writes and secondary replicas exist to take the place of a failed primary replica by means of a leader election. So far, so good. The problems arise once we start demanding high write throughput from our fault-tolerant MongoDB cluster.

The Effectively-Global Lock

In a replicated cluster, MongoDB will lock at least two databases for every write: the database containing the document being modified, and the 'local' database. The 'local' database contains the operations log. Secondary replicas tail the oplog in order to replicate writes.

This means that, for a given replica set, every write goes through what is effectively a global lock. Page-level locking should help, but not much, since it is expected that subsequent entries in the oplog will usually reside on the same page.

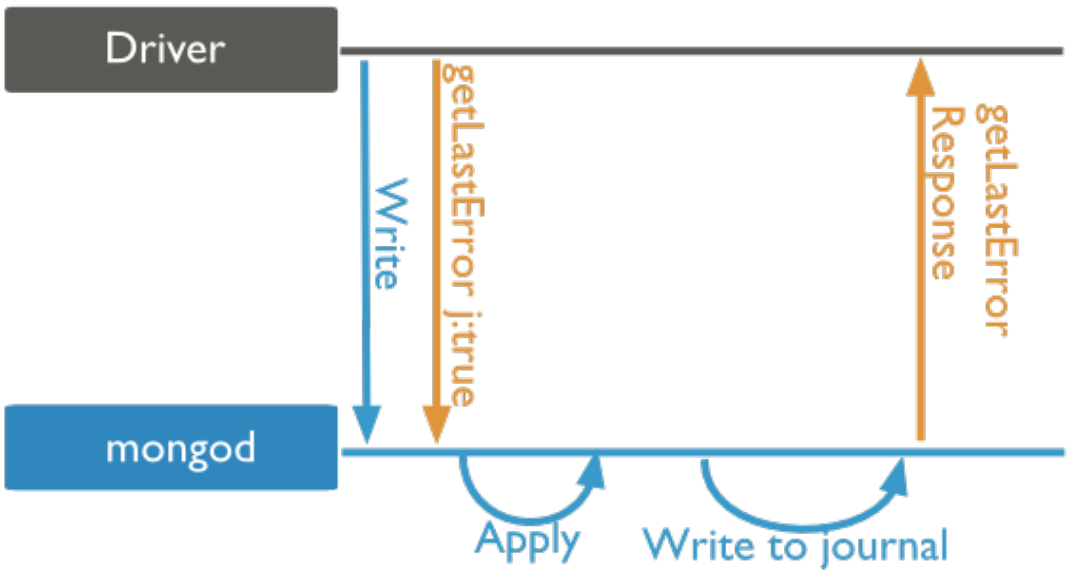
Journal ≠ OpLog?

Why is there a journal as well as the oplog, anyway? The two should contain semantically the same information. The OpLog is also journaled, so the oplog journal is a journal journal. That means there are four writes for every database operation:

1. Journal OpLog Write
2. Apply OpLog Write (operation is now available for replication to secondaries)

3. Journal Target Write
4. Apply Target Write

This diagram from the MongoDB docs doesn't inspire much faith:

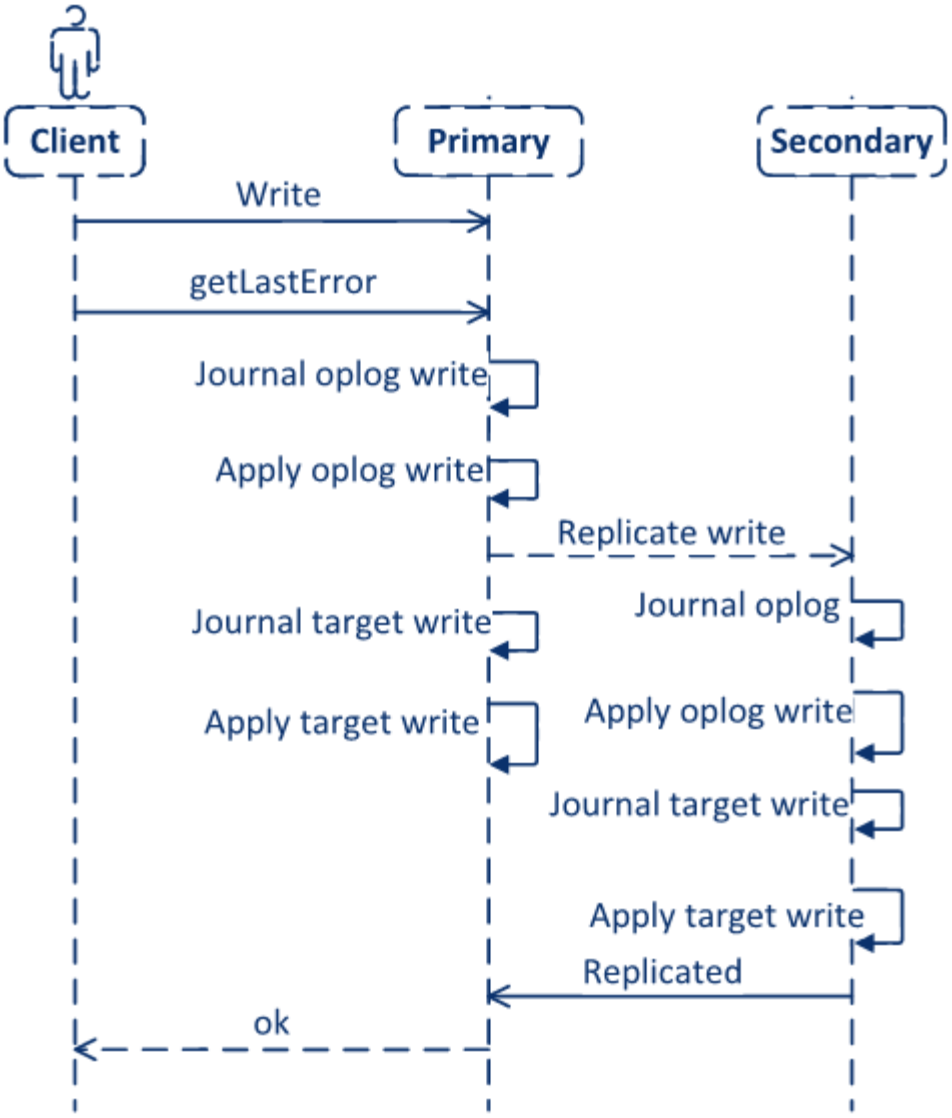


Apply operation and then journal that thing we just did... What could go wrong?

I believe that image is misleading, but by applying an operation before journaling it, the database can be easily corrupted in the case that a crash occurs before journaling completes.

Replicate via Disk

There is a more subtle implication of this architecture: writes have to hit the primary's disk before they are eligible for replication to quorum members. In a cloud computing scenario, this likely means travelling over the network to another machine's disk, increasing latency even more. By using a database for replication, rather than doing it in-memory, disk access has to be serialized across machines. Let me visualize that for you. I welcome corrections to the following:



For simplicity, only one secondary is shown.

No "Replica Journaled" Write Concern Level

MongoDB uses a "writeconcern" parameter in order to ensure that durability requirements are met on a per-operation basis. The following write concern levels exist:

- **Unacknowledged** – Fire & Forget mode. Return immediately. For when you truly don't care.
- **Acknowledged** – Return when the primary has seen the operation. It will probably be made durable.
- **Journaled** – Return when the primary has committed a journal entry describing the change. The current primary has at least made the operation durable.

- **Replica Acknowledged** – Return when a certain number of replicas (including the primary) has seen the change. It's unlikely that all replicas will crash immediately after acknowledging, right?

What's missing is a "Replica Journaled" write concern level, the one where a quorum of replicas have persisted the operation to storage. A DC power outage should never result in corrupted data. Think of a distributed system as always being in a partially degraded state and designing for this, because any sufficiently large system will exhibit that trait.

Recommendations

- Either use the OpLog as a journal or use the other journal as a journal and replicate that. Less writes and less failure modes.
- Replicate candidate operations and persist them concurrently on each replica, rather than first on the primary and then on the secondaries.
- Use a distributed consensus algorithm like Paxos or Raft to agree on a journal which is persisted by each replica.
- Provide a truly durable write consistency level. Make this the safe & sane default.

This should result in a faster and safer system which is easier to reason about. It would make a lot of sense to leverage an existing framework for distributed consensus rather than baking a new one. I'd happily accept corrections to anything written here. There is no pride in pointing out issues with other people's work, these are just observations my manager & I made while writing a real-time metrics monitoring system at Microsoft. We dropped MongoDB in favour of Azure Table Service and never looked back. Azure Table Service is implemented atop a journal, like Cassandra, which is great for write throughput. We structured our RowKeys and PartitionKeys in a way which gave us sufficient range query semantics for such a system. There's more to this story which I might write about in a few months time.

TL;DR: Unify the oplog and the journal, use distributed consensus for agreeing on its contents, and provide a safer clustered write concern level.

Good reads:

- The Log: What every software engineer should know about real-time data's unifying abstraction
- Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency (Video here)

Share this:

Twitter Reddit Facebook Google Email More

Related

Azure DocumentDB Primer
August 22, 2014
In "azure"

Service Fabric
April 30, 2015
In "cloud"

Microsoft Orleans - Why you should care
April 3, 2014
Similar post

Posted in Uncategorized