How We Designed and Model-Checked MongoDB Reconfiguration Protocol

Siyuan Zhou
Lead Engineer, MongoDB

MongoDB Replication

MongoDB ensures fault tolerance with a **Raft**-like consensus protocol.

- All nodes in a **replica set** store the same data.
- A **Primary** writes into the **oplog** and **secondaries** replicate newer oplog entries.
- When no primary exists, a secondary can run an **election** for a higher **term**.
- A secondary becomes a **primary** by collecting votes from a **majority** of nodes, so only one can become a primary in a given term.
- An oplog entry is **committed** when replicated to a **majority** of nodes within a primary's term.
- Safety guarantee: Committed writes will be safe even if some nodes fail

```
{
    "t": 1,                                          // Term of the primary
    "ts": Timestamp(1615262400, 1),                  // Timestamp of the write
    "op": "i",                                       // Insert
    "ns": "myblog.posts",                            // Collection namespace
    "o": {                                           // The document to insert
        "_id": ObjectId("5c1a3b1234567890abcdef12"),
        "title": "Hello TLA+ Conf!",
        "content": "Let's talk about TLA+ and MongoDB reconfig."
    }
    ...
}
```

Replication Reconfiguration

- A <u>configuration / config</u> defines the membership of a replica set.
- Necessary to add or remove nodes via <u>**reconfiguration / reconfig**</u>.
- Consensus correctness depends on “majority”, but when adding or removing nodes, the definition of “majority” is changing!
- Challenge: Ensuring system correctness during reconfiguration.
  - Notoriously hard to design.
  - A critical safety bug in one of Raft reconfig protocols was found after initial publication,

Legacy MongoDB Reconfig Protocol

- Gossip protocol via heartbeats
  - Only the primary can run reconfig
  - Each configuration has a user-defined config version
  - A node installs higher config immediately upon learning
- Unsafe in certain cases
  - Need for a new safe reconfig protocol
- Supports “force reconfig”
  - Any node can install a new config even if majority of nodes are offline and no primary exists
  - Get the application back knowing the risk of losing some most recent data
  - A feature needed by on-prem customers

New Reconfig Protocol

- Initial design to adopt Raft's log-based reconfig protocol
  - Incompatible with "force reconfig"
  - Require both log-based and gossip-based implementations
  - Complex upgrade / downgrade
- Goal: Develop a heartbeat reconfig protocol supporting "force reconfig" with minimal changes

Inspiration from Raft

Can we adopt Raft's simple safety rules for reconfig?

- Only single-node changes are allowed at each time
  - e.g., adding one node is allowed, but adding two at the same time isn't.
- New config is only accepted when the previous config is committed

How to guarantee the correctness of the new protocol?

- Leverage TLA+ and model checking
- The team had TLA+ experiences on smaller problems in 2019

Day 1 - Initial Attempt on Single Node Change

- Add two reconfig related actions in TLA+ spec:
  - **Reconfig**: sends the config to the primary and installs it immediately.
  - **SendConfig**: gossips a new config with a higher config version to another node.
- Model the rule of single node change
  - Any majority of adjacent configs always overlaps with each other.

```
[X] [X] [ ]         (The majority of 3 is 2)

[ ] [X] [X] [X]     (The majority of 4 is 3)
```

- Reproduced a known bug with just 150 lines of code change

```
\* A reconfig occurs on node i.
Reconfig(i) ==
   \* Pick some arbitrary subset of servers to reconfig to.
   \* “Server” is the set of all nodes, e.g., {n1, n2, n3}.
   \E newConfig \in SUBSET Server :
       \* The node must currently be a leader.
       /\ state[i] = Primary
       \* Add or remove a single node.
       /\ \/ Cardinality(config[i]) + 1 = Cardinality(newConfig)
          \/ Cardinality(config[i]) - 1 = Cardinality(newConfig)
       \* Make sure to include this node in the new config.
       /\ i \in newConfig
       \* The config on this node takes effect immediately.
       ...
```

Day 1 - Safety Properties for Model Checking

- **ElectionSafety**: Never elect two primaries in the same term.
- **NeverRollbackCommitted**: Never roll back committed oplog entries.

Both n1 and n2 are primaries, but in different terms

state	config	current term	config version
n1 :> Primary	n1 :> {n1, n2, n3}	**n1 :> 1**	n1 :> 0
n2 :> Primary	n2 :> {n1, n2, n3}	**n2 :> 2**	n2 :> 0
n3 :> Secondary	n3 :> {n1, n2, n3}	n3 :> 2	n3 :> 0

n1 removes one node			
state	**config**	**current term**	**config version**
n1 :> Primary	**n1 :> {n1, n2}**	**n1 :> 1**	**n1 :> 1**
n2 :> Primary	n2 :> {n1, n2, n3}	**n2 :> 2**	n2 :> 0
n3 :> Secondary	n3 :> {n1, n2, n3}	n3 :> 2	n3 :> 0

n1 removes one more node

state	config	current term	config version
n1 :> Primary	**n1 :> {n1}**	**n1 :> 1**	**n1 :> 2**
n2 :> Primary	n2 :> {n1, n2, n3}	**n2 :> 2**	n2 :> 0
n3 :> Secondary	n3 :> {n1, n2, n3}	n3 :> 2	n3 :> 0

Inspiration from Raft

Can we adopt Raft's simple safety rules for reconfig?

- Only single-node changes is allowed
  - e.g., adding one node is allowed, but adding two isn't.
- New config is only accepted when the previous config is committed

Day 2 / Day 3 - Efforts on Config Commitment

After a few iterations, we added the following rules for the Reconfig action:

- (TermQuorumCheck) Check the primary is still valid by comparing its term with a majority of nodes.
- (ConfigQuorumCheck) Check a majority of nodes have the same config version as the primary.

```
\* Am I talking to a quorum as primary?
TermQuorumCheck(self, s) == currentTerm[self] >= currentTerm[s]
\* Have a quorum of nodes received my config?
ConfigQuorumCheck(self, s) == configVersion[self] = configVersion[s]
ConfigIsSafe(i) == /\ \E q \in Quorums(config[i]):
                       \A s \in q : /\ TermQuorumCheck(i, s)
                                    /\ ConfigQuorumCheck(i, s)
\* A reconfig occurs on node i.
Reconfig(i) ==
    \E newConfig \in SUBSET Server :
        /\ state[i] = Primary
        \* Only allow a new config to be installed if the current config is "safe".
        /\ ConfigIsSafe(i)
        \* Add or remove a single node.
        /\ \/ \E n \in newConfig : newConfig \ {n} = config[i]  \* add 1.
           \/ \E n \in config[i] : config[i] \ {n} = newConfig  \* remove 1.
        /\ i \in newConfig
        ...
```

Day 4 - Oplog and Config Dependencies

- We found a counterexample around the dependency between log and config.
- Raft orders configs and log entries implicitly.
- The heartbeat reconfig protocol lost this implicit dependency.
- Reproduced with model checking.

Primary commits a write with config {n1, n2, n3}

log	**state**	**config**
n1 :> **<<[term |-> 1]>>**	**n1 :> Primary**	**n1 :> {n1, n2, n3}**
n2 :> **<<[term |-> 1]>>**	n2 :> Secondary	**n2 :> {n1, n2, n3}**
n3 :> <<>>	n3 :> Secondary	**n3 :> {n1, n2, n3}**
n4 :> <<>>	n4 :> Secondary	n4 :> {n1, n2, n3}
n5 :> <<>>	n5 :> Secondary	n5 :> {n1, n2, n3}

Primary adds n4 to the config and propagates the config

<table>
<tr><th>log</th><th>state</th><th>config</th></tr>
<tr><td>n1 :> **<<[term |-> 1]>>**</td><td>**n1 :> Primary**</td><td>**n1 :> {n1, n2, n3, n4}**</td></tr>
<tr><td>n2 :> **<<[term |-> 1]>>**</td><td>n2 :> Secondary</td><td>**n2 :> {n1, n2, n3, n4}**</td></tr>
<tr><td>n3 :> <<>></td><td>n3 :> Secondary</td><td>**n3 :> {n1, n2, n3, n4}**</td></tr>
<tr><td>n4 :> <<>></td><td>n4 :> Secondary</td><td>**n4 :> {n1, n2, n3, n4}**</td></tr>
<tr><td>n5 :> <<>></td><td>n5 :> Secondary</td><td>n5 :> {n1, n2, n3}</td></tr>
</table>

Primary adds n5 to the config and propagates the config.

<table>
<tr><th>log</th><th>state</th><th>config</th></tr>
<tr><td>n1 :> **<<[term |-> 1]>>**</td><td>**n1 :> Primary**</td><td>**n1 :> {n1, n2, n3, n4, n5}**</td></tr>
<tr><td>n2 :> **<<[term |-> 1]>>**</td><td>n2 :> Secondary</td><td>**n2 :> {n1, n2, n3, n4, n5}**</td></tr>
<tr><td>n3 :> <<>></td><td>n3 :> Secondary</td><td>**n3 :> {n1, n2, n3, n4, n5}**</td></tr>
<tr><td>n4 :> <<>></td><td>n4 :> Secondary</td><td>**n4 :> {n1, n2, n3, n4, n5}**</td></tr>
<tr><td>n5 :> <<>></td><td>n5 :> Secondary</td><td>**n5 :> {n1, n2, n3, n4, n5}**</td></tr>
</table>

n3 becomes the primary and commits a new write.

n1 and n2 will rollback.

log	state	config
n1 :> <<[term |-> 1]>>	n1 :> Secondary	**n1 :> {n1, n2, n3, n4, n5}**
n2 :> <<[term |-> 1]>>	n2 :> Secondary	**n2 :> {n1, n2, n3, n4, n5}**
n3 :> **<<[term |-> 2]>>**	n3 :> **Primary**	**n3 :> {n1, n2, n3, n4, n5}**
n4 :> **<<[term |-> 2]>>**	n4 :> Secondary	**n4 :> {n1, n2, n3, n4, n5}**
n5 :> **<<[term |-> 2]>>**	n5 :> Secondary	**n5 :> {n1, n2, n3, n4, n5}**

n3 becomes the primary and commits a new write.

n1 and n2 will rollback.

<table>
<tr><th>log</th><th>state</th><th>config</th></tr>
<tr><td>n1 :> <<[term |-> 1]>></td><td>n1 :> Secondary</td><td>n1 :> {n1, n2, n3, n4, n5}</td></tr>
<tr><td>n2 :> <<[term |-> 1]>></td><td>n2 :> Secondary</td><td>n2 :> {n1, n2, n3, n4, n5}</td></tr>
<tr><td>n3 :> <<[term |-> 2]>></td><td>n3 :> Primary</td><td>n3 :> {n1, n2, n3, n4, n5}</td></tr>
<tr><td>n4 :> <<[term |-> 2]>></td><td>n4 :> Secondary</td><td>n4 :> {n1, n2, n3, n4, n5}</td></tr>
<tr><td>n5 :> <<[term |-> 2]>></td><td>n5 :> Secondary</td><td>n5 :> {n1, n2, n3, n4, n5}</td></tr>
</table>

When adding n5, the oplog entry committed in 3-node config, hasn't been committed in 4-node config.

```
log                       state             config

n1 :> <<[term |-> 1]>>    n1 :> Primary     n1 :> {n1, n2, n3, n4}
n2 :> <<[term |-> 1]>>    n2 :> Secondary   n2 :> {n1, n2, n3, n4}
n3 :> <<>>                n3 :> Secondary   n3 :> {n1, n2, n3, n4}
n4 :> <<>>                n4 :> Secondary   n4 :> {n1, n2, n3, n4}
n5 :> <<>>                n5 :> Secondary   n5 :> {n1, n2, n3}
```

```
\* Can the last op be committed in the current config of node i?
\*
\* CommitEntry() is to commit the last log entry on a primary when the entry is
\* replicated to a majority of nodes in its term, according to its current config.
OpCommittedInConfig(primary) == ENABLED CommitEntry(primary)

\* Is the config on node i currently "safe"?
ConfigIsSafe(i) ==
    /\ \E q \in Quorums(config[i]):
        \A s \in q : /\ TermQuorumCheck(i, s)
                     /\ ConfigQuorumCheck(i, s)
    /\ OpCommittedInConfig(i)
```

Day 5 - Config Consensus Counterexample

- Model checker found another counterexample after running for about one day

N1 is the primary and removes a node.

state	config	current term	config version
n1 :> Primary	**n1 :> {n1, n2, n3}**	**n1 :> 1**	**n1 :> 1**
n2 :> Secondary	n2 :> {n1, n2, n3, n4}	n2 :> 1	n2 :> 0
n3 :> Secondary	n3 :> {n1, n2, n3, n4}	n3 :> 1	n3 :> 0
n4 :> Secondary	n4 :> {n1, n2, n3, n4}	n4 :> 0	n4 :> 0

N2 becomes the primary and removes a different node.

state	config	current term	config version
n1 :> Primary	**n1 :> {n1, n2, n3}**	**n1 :> 1**	**n1 :> 1**
n2 :> Primary	**n2 :> {n1, n2, n4}**	**n2 :> 2**	**n2 :> 1**
n3 :> Secondary	n3 :> {n1, n2, n3, n4}	n3 :> 2	n3 :> 0
n4 :> Secondary	n4 :> {n1, n2, n3, n4}	n4 :> 2	n4 :> 0

N1 propagates its config but steps down on seeing higher term.

state	**config**	**current term**	**config version**
n1 :> Secondary	**n1 :> {n1, n2, n3}**	**n1 :> 2**	**n1 :> 1**
n2 :> Primary	**n2 :> {n1, n2, n4}**	**n2 :> 2**	**n2 :> 1**
n3 :> Secondary	**n3 :> {n1, n2, n3}**	n3 :> 2	**n3 :> 1**
n4 :> Secondary	n4 :> {n1, n2, n3, n4}	n4 :> 2	n4 :> 0

N1 becomes primary again in term 3

state	config	current term	config version
n1 :> Primary	**n1 :> {n1, n2, n3}**	**n1 :> 3**	**n1 :> 1**
n2 :> Primary	**n2 :> {n1, n2, n4}**	**n2 :> 2**	**n2 :> 1**
n3 :> Secondary	**n3 :> {n1, n2, n3}**	**n3 :> 3**	**n3 :> 1**
n4 :> Secondary	n4 :> {n1, n2, n3, n4}	n4 :> 2	n4 :> 0

N2 propagates its config.

state	config	current term	config version
n1 :> Primary	n1 :> {n1, n2, n3}	**n1 :> 3**	**n1 :> 1**
n2 :> Primary	n2 :> {n1, n2, n4}	**n2 :> 2**	**n2 :> 1**
n3 :> Secondary	n3 :> {n1, n2, n3}	**n3 :> 3**	**n3 :> 1**
n4 :> Secondary	**n4 :> {n1, n2, n4}**	n4 :> 2	**n4 :> 1**

Day 5 - Config Consensus

- On a new reconfig, we need to ensure in the future, no primaries would ever be elected in earlier configs, so that earlier configs are "deactivated".
- This means that if two configs "compete", they differ by at most one server.
- **Agreeing on the config among the nodes is a consensus problem!**
    - Separate from the oplog consensus

Day 5 - Config Consensus

- Config consensus and oplog consensus share similarities.
  - ClientRequest => Reconfig
  - GetEntry & RollbackEntries => SendConfig
- Identify and order configs with <config term, config version>
  - Like an oplog entry is defined and ordered by <entry term, entry timestamp>.
- Merge both elections by adding a rule of comparing configs' terms and versions.
- Borrow the definition of "commitment" from the oplog consensus.
- Rewrite the current config with the latest term on winning elections.

Optimizing the Spec

- Removed unnecessary states, like the voting states.
- Fine-tuned initial configuration to focus on interesting states.
- Aligned with implementation, e.g., term propagation.
- Only focus on ensuring the Election Safety property.
    - No two primaries can be elected in the same term.
- Faster model checking with larger models.
- Added an action to simulate a shutdown at any time
    - Significantly expanding the state space.

Checking Liveness

- Combining the elections restricts the behavior
- Shall we allow config propagation without a primary?
    - Raft cannot propagate oplog withtout a primary, but MongoDB date replication can.
    - Allowed in reconfig protocol.
- Attempted to avoid seemingly unnecessary complexity, but found liveness issues.

Conclusions

- We designed and model checked a new reconfig protocol with TLA+.
  - 4+ iterations in the first few hours.
  - Got a draft protocol in one week.
  - Within two weeks, we finalized the protocol for safety and liveness.
- The scope of the implementation change became much smaller.
  - Delivered the project in three months with three to four developers
- "Force reconfig" is implemented using the same mechanism with relaxed rules.
- Simpler upgrade / downgrade.
- Correctness proof and formal verification were done by William Schultz.
- Reliable in production since MongoDB 4.4 released in 2019.

Takeaways

- Model checking is a great tool to answer "what if" questions for fast iteration.
- Model checking helped us to reason about the system critically and quickly.
- No bugs found in the protocol, but found bugs in the implementation.
  - E.g. atomic step-down action in spec requires multiple lock acquisitions and database writes.
  - Covered by unit tests and integration tests
- Safety isn’t guaranteed beyond the TLA+ spec.
  - “Force reconfig” isn’t modeled and is still unsafe, but it’s only for on-prem customers and never used on MongoDB Atlas - our hosted MongoDB as a service.

References & Credits

- Design and Analysis of a Logless Dynamic Reconfiguration Protocol
  - William Schultz, Siyuan Zhou, Ian Dardik, Stavros Tripakis
- Formal Verification of a Distributed Dynamic Reconfiguration Protocol
  - William Schultz, Ian Dardik, Stavros Tripakis
- The TLA+ spec and its Git history can be found on Github
- The latest version of the TLA+ spec is in the MongoDB repository
- Collaborated with William Schultz and Tess Avitabile on the design
- Credits to the MongoDB Replication team for the implementation

Thanks!

Model Checking Stats

	ElectionSafety	NeverRollbackCommitted
Number of servers	5	4
Max oplog length	-	2
Max config versions	4	3
Max terms	4	3
Distinct states	812,587,401	345,587,274
Duration	19h 28min	8h 06min