Design and Analysis of a Logless Dynamic Reconfiguration Protocol

William Schultz
Northeastern University, Boston, MA, USA

Siyuan Zhou
MongoDB, New York, NY, USA

Ian Dardik
Northeastern University, Boston, MA, USA

Stavros Tripakis
Northeastern University, Boston, MA,USA

Abstract

Distributed replication systems based on the replicated state machine model have become ubiquitous as the foundation of modern database systems. To ensure availability in the presence of faults, these systems must be able to dynamically replace failed nodes with healthy ones via *dynamic reconfiguration*. MongoDB is a document oriented database with a distributed replication mechanism derived from the Raft protocol. In this paper, we present *MongoRaftReconfig*, a novel dynamic reconfiguration protocol for the MongoDB replication system. *MongoRaftReconfig* utilizes a logless approach to managing configuration state and decouples the processing of configuration changes from the main database operation log. The protocol's design was influenced by engineering constraints faced when attempting to redesign an unsafe, legacy reconfiguration mechanism that existed previously in MongoDB. We provide a safety proof of *MongoRaftReconfig*, along with a formal specification in TLA+. To our knowledge, this is the first published safety proof and formal specification of a reconfiguration protocol for a Raft-based system. We also present results from model checking the safety properties of *MongoRaftReconfig* on finite protocol instances. Finally, we discuss the conceptual novelties of *MongoRaftReconfig*, how it can be understood as an optimized and generalized version of the single server reconfiguration algorithm of Raft, and present an experimental evaluation of how its optimizations can provide performance benefits for reconfigurations.

2012 ACM Subject Classification Information systems → Parallel and distributed DBMSs; Software and its engineering → Software verification

Keywords and phrases Fault Tolerance, Dynamic Reconfiguration, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.26

Related Version *Full Version*: `https://arxiv.org/abs/2102.11960` [28]

Supplementary Material *Software (TLA+ specifications [26])*: `https://doi.org/10.5281/zenodo.5715510`

Funding This work has been partially supported by NSF award CNS-1801546.

Acknowledgements We would like to thank Tess Avitabile for her critical insights during the development of the reconfiguration protocol and discovery of subtle bugs in early design proposals. We would like to thank Judah Schvimer, A. Jesse Jiryu Davis, Pavi Vetriselvan, and Ali Mir for offering helpful insights during the protocol design and implementation process. We would also like to thank Shuai Mu for providing helpful comments on initial drafts of this paper.

1 Introduction

Distributed replication systems based on the replicated state machine model [24] have become ubiquitous as the foundation of modern, fault-tolerant data storage systems. In order for these systems to ensure availability in the presence of faults, they must be able to dynamically replace failed nodes with healthy ones, a process known as *dynamic reconfiguration*. The protocols for building distributed replication systems have been well studied and implemented in a variety of systems [4, 7, 9, 30]. Paxos [12] and, more recently, Raft [22], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [1] that has not been explored as extensively as the foundational consensus protocols underlying these systems. Variants of Paxos have examined the problem of dynamic reconfiguration but these reconfiguration techniques may require changes to a running system that impact availability [14] or require the use of an external configuration master [15]. The Raft consensus protocol, originally published in 2014, provided a dynamic reconfiguration algorithm in its initial publication, but did not include a precise discussion of its correctness or include a formal specification or proof. A critical safety bug [20] in one of its reconfiguration protocols was found after initial publication, demonstrating that the design and verification of reconfiguration protocols for these systems is a challenging task.

MongoDB [17] is a general purpose, document oriented database which implements a distributed replication system [27] for providing high availability and fault tolerance. MongoDB's replication system uses a novel consensus protocol that derives from Raft [34]. Since its inception, the MongoDB replication system has provided a custom, legacy protocol for dynamic reconfiguration of replica members that was not based on a published algorithm. This legacy protocol managed configurations in a *logless* fashion i.e. each server only stored its latest configuration. In addition, it decoupled reconfiguration processing from the main database operation log. These features made for a simple and appealing protocol design, and it was sufficient to provide basic reconfiguration functionality to clients. The legacy protocol, however, was known to be unsafe in certain cases. In recent versions of MongoDB, reconfiguration has become a more common operation, necessitating the need for a redesigned, safe reconfiguration protocol with rigorous safety guarantees. From a system engineering perspective, a primary goal was to keep design and implementation complexity low. Thus, it was desirable that the new reconfiguration protocol minimize changes to the legacy protocol to the extent possible. In this paper, we present *MongoRaftReconfig*, a novel dynamic reconfiguration protocol that achieves the above design goals.

MongoRaftReconfig provides safe, dynamic reconfiguration, utilizes a logless approach to managing configuration state, and decouples reconfiguration processing from the main database operation log. Thus, it bears a high degree of architectural and conceptual similarity to the legacy MongoDB protocol, satisfying our original design goal of minimizing changes to the legacy protocol. We provide rigorous safety guarantees of *MongoRaftReconfig*, including a proof of the protocol's main safety properties along with a formal specification in TLA+ [16], a specification language for describing distributed and concurrent systems. To our knowledge, this is the first published safety proof and formal specification of a reconfiguration protocol for a Raft-based system. We also verified the safety properties of finite instances of *MongoRaftReconfig* using the TLC model checker [33], which provides additional confidence in its correctness. Finally, we discuss the conceptual novelties of *MongoRaftReconfig*, related to its logless design and decoupling of reconfiguration processing. In particular, we discuss how it can be understood as an optimized and generalized variant of

the single server Raft reconfiguration protocol. We also include a preliminary experimental evaluation of how these optimizations can provide performance benefits over standard Raft, by allowing reconfigurations to bypass the main operation log.

To summarize, in this paper we make the following contributions:

- We present *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol for the MongoDB replication system.
- We present a proof of *MongoRaftReconfig*'s key safety properties. To our knowledge, this is the first published safety proof of a reconfiguration protocol for a Raft-based system.
- We present a formal specification of *MongoRaftReconfig* in TLA+. To our knowledge, this is the first published formal specification of a reconfiguration protocol for a Raft-based system.
- We present results of model checking the safety properties of *MongoRaftReconfig* on finite protocol instances using the TLC model checker.
- We discuss the conceptual novelties of *MongoRaftReconfig*, and how it can be understood as an optimized and generalized variant of the single server Raft reconfiguration protocol.
- We provide a preliminary experimental evaluation of *MongoRaftReconfig*'s performance benefits, demonstrating how it improves upon reconfiguration in standard Raft.

2 Background

2.1 System Model

Throughout this paper, we consider a set of *server* processes $Server = \{s_1, s_2, ..., s_n\}$ that communicate by sending messages. We assume an asynchronous network model in which messages can be arbitrarily dropped or delayed. We assume servers can fail by stopping but do not act maliciously i.e. we assume a "fail-stop" model with no Byzantine failures. We define both a *member set* and a *quorum* as elements of 2^{Server}. Member sets and quorums have the same type but refer to different conceptual entities. For any member set m, and any two non-empty member sets m_i, m_j, we define the following:

$$Quorums(m) \triangleq \{s \in 2^m : |s| \cdot 2 > |m|\} \tag{1}$$

$$QuorumsOverlap(m_i, m_j) \triangleq \forall q_i \in Quorums(m_i), q_j \in Quorums(m_j) : q_i \cap q_j \neq \emptyset \tag{2}$$

where $|S|$ denotes the cardinality of a set S. We refer to Definition 2 as the *quorum overlap* condition.

2.2 Raft

Raft [19] is a consensus protocol for implementing a replicated log in a system of distributed servers. It has been implemented in a variety of systems across the industry [21]. Throughout this paper, we refer to the original Raft protocol as described and specified in [19] as *standard Raft*.

The core Raft protocol implements a replicated state machine using a static set of servers. In the protocol, time is divided into *terms* of arbitrary length, where terms are numbered with consecutive integers. Each term has at most one leader, which is selected via an *election* that occurs at the beginning of a term. To dynamically change the set of servers operating the protocol, Raft includes two, alternate algorithms: *single server membership change* and *joint consensus*. In this paper we are only concerned with *single server membership change*. The single server change approach aims to simplify reconfiguration by allowing only reconfigurations that add or remove a single server. Reconfiguration is accomplished by

writing a special reconfiguration entry into the main Raft operation log that alters the local configuration of a server. In this paper, when referring to reconfiguration in standard Raft, we assume it to mean the single server change protocol.

2.3 Replication in MongoDB

MongoDB is a general purpose, document oriented database that stores data in JSON-like objects. A MongoDB database consists of a set of collections, where a collection is a set of unique documents. To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB servers that act as a consensus group, where each server maintains a logical copy of the database state.

MongoDB replica sets utilize a replication protocol that is derived from Raft, with some extensions. We refer to MongoDB's abstract replication protocol, without dynamic reconfiguration, as *MongoStaticRaft*. This protocol can be viewed as a modified version of standard Raft that satisfies the same underlying correctness properties. A more in depth description of *MongoStaticRaft* is given in [34, 27], but we provide a high level overview here, since the *MongoRaftReconfig* reconfiguration protocol is built on top of *MongoStaticRaft*. In a replica set running *MongoStaticRaft* there exists a single *primary* server and a set of *secondary* servers. As in standard Raft, there is a single primary elected per term. The primary server accepts client writes and inserts them into an ordered operation log known as the *oplog*. The oplog is a logical log where each entry contains information about how to apply a single database operation. Each entry is assigned a monotonically increasing timestamp, and these timestamps are unique and totally ordered within a server log. These log entries are then replicated to secondaries which apply them in order leading to a consistent database state on all servers. When the primary learns that enough servers have replicated a log entry in its term, the primary will mark it as *committed*, guaranteeing that the entry is permanently durable in the replica set. Clients of the replica set can issue writes with a specified *write concern* level, which indicates the durability guarantee that must be satisfied before the write can be acknowledged to the client. Providing a write concern level of *majority* ensures that a write will not be acknowledged until it has been marked as committed in the replica set. A key, high level safety requirement of the replication protocol is that if a write is acknowledged as committed to a client, it should be durable in the replica set.

3 MongoRaftReconfig: A Logless Dynamic Reconfiguration Protocol

In this section we present the *MongoRaftReconfig* dynamic reconfiguration protocol. First, we provide an overview and some intuition on the protocol design in Section 3.1. Section 3.2 provides a high level, informal description of the protocol along with a condensed pseudocode description in Algorithm 1. Sections 3.3 and 3.4 provide additional detail on the mechanisms required for the protocol to operate safely, and the TLA+ formal specification of *MongoRaftReconfig* is discussed briefly in Section 3.5.

The complete description of *MongoRaftReconfig* is left to the full version of the paper [28]. The pseudocode presented in Algorithm 1 describes the reconfiguration specific behaviors of *MongoRaftReconfig*, which are the novel aspects of the protocol and the contributions of this paper.

3.1 Overview and Intuition

Dynamic reconfiguration allows the set of servers operating as part of a replica set to be modified while maintaining the core safety guarantees of the replication protocol. Many consensus based replication protocols [29, 14, 22] utilize the main operation log (the *oplog*, in MongoDB) to manage configuration changes by writing special reconfiguration log entries. The *MongoRaftReconfig* protocol instead decouples configuration updates from the main operation log by managing the configuration state of a replica set in a separate, logless replicated state machine, which we refer to as the *config state machine*. The config state machine is maintained alongside the oplog, and manages the configuration state used by the overall protocol.

In order to ensure safe reconfiguration, *MongoRaftReconfig* imposes specific restrictions on how reconfiguration operations are allowed to update the configuration state of the replica set. First, it imposes a *quorum overlap* condition on any reconfiguration from C to C', which is an approach adopted from the Raft single server reconfiguration algorithm. This ensures that all quorums of two adjacent configurations overlap with each other, and so can safely operate concurrently. In order to allow the system to pass through many configurations over time, though, *MongoRaftReconfig* imposes additional restrictions which address two essential aspects required for safe dynamic reconfiguration: (1) *deactivation* of old configurations and (2) *state transfer* from old configurations to new configurations. Essentially, it must ensure that old configurations, which may not overlap with newer configurations, are appropriately prevented from executing disruptive operations (e.g. electing a primary or committing a write), and it must also ensure that relevant protocol state from old configurations is properly transferred to newer configurations before they become active. The details of these restrictions and their safety implications are discussed further in Section 3.3.

In the remainder of this section we give an overview of the behaviors of *MongoRaftReconfig*, along with a pseudocode description of the protocol. We discuss its correctness in more depth in Section 4.

3.2 High Level Protocol Behavior

At a high level, dynamic reconfiguration in *MongoRaftReconfig* consists of two main aspects: (1) updating the current configuration and (2) propagating new configurations between servers. Configurations also have an impact on election behavior which we discuss below, in Section 3.4. Formally, a *configuration* is defined as a tuple (m, v, t), where $m \in 2^{Server}$ is a member set, $v \in \mathbb{N}$ is a numeric configuration *version*, and $t \in \mathbb{N}$ is the numeric *term* of the configuration. For convenience, we refer to the elements of a configuration tuple $C = (m, v, t)$ as, respectively, $C.m$, $C.v$ and $C.t$. Each server of a replica set maintains a single, durable configuration, and it is assumed that, initially, all nodes begin with a common configuration, $(m_{init}, 1, 0)$, where $m_{init} \in (2^{Server} \setminus \emptyset)$.

To update the current configuration of a replica set, a client issues a *reconfiguration* command to a primary server with a new, desired configuration, C'. Reconfigurations can only be executed on primary servers, and they update the primary's current local configuration C to the specified configuration C'. The version of the new configuration, $C'.v$, must be greater than the version of the primary's current configuration, $C.v$, and the term of C' is set equal to the current term of the primary processing the reconfiguration. After a reconfiguration has occurred on a primary, the updated configuration needs to be communicated to other servers in the replica set. This is achieved in a simple, gossip like manner. Secondaries receive information about the configurations of other servers via periodic heartbeats. They need to

Algorithm 1 Pseudocode description of *MongoRaftReconfig* reconfiguration specific behavior.

Definitions

$C_{(i)} \triangleq (config[i], configVersion[i], configTerm[i])$
$C_i > C_j \triangleq (C_i.t > C_j.t) \vee (C_i.t = C_j.t \wedge C_i.v > C_j.v)$
$C_i \geq C_j \triangleq (C_i > C_j) \vee ((C_i.v, C_i.t) = (C_j.v, C_j.t))$
$Q1(i) \triangleq \exists Q \in Quorums(config[i]) : \forall j \in Q : (C_{(j)}.v, C_{(j)}.t) = (C_{(i)}.v, C_{(i)}.t)$ ▷ Config Quorum Check
$Q2(i) \triangleq \exists Q \in Quorums(config[i]) : \forall j \in Q : term[j] = term[i]$ ▷ Term Quorum Check
$P1(i) \triangleq \exists Q \in Quorums(config[i])$: all entries committed in terms $\leq term[i]$ are committed in Q

1: **State and Initialization**
2: Let $m_{init} \in 2^{Server} \setminus \emptyset$
3: $\forall i \in Server$:
4: $term[i] \in \mathbb{N}$, initially 0
5: $state[i] \in \{Pri., Sec.\}$, initially *Secondary*
6: $config[i] \in 2^{Server}$, initially m_{init}
7: $configVersion[i] \in \mathbb{N}$, initially 1
8: $configTerm[i] \in \mathbb{N}$, initially 0
9:
10: **Actions**
11: **action:** RECONFIG(i, m_{new})
12: **require** $state[i] = Primary$
13: **require** $Q1(i) \wedge Q2(i) \wedge P1(i)$
14: **require** $QuorumsOverlap(config[i], m_{new})$
15: $config[i] \leftarrow m_{new}$
16: $configVersion[i] \leftarrow configVersion[i] + 1$
17:
18: **action:** SENDCONFIG(i, j)
19: **require** $state[j] = Secondary$
20: **require** $C_{(i)} > C_{(j)}$
21: $C_{(j)} \leftarrow C_{(i)}$
22:
23: **action:** BECOMELEADER(i, Q)
24: **require** $Q \in Quorums(config[i])$
25: **require** $i \in Q$
26: **require** $\forall v \in Q : C_{(i)} \geq C_{(v)}$
27: **require** $\forall v \in Q : term[i] + 1 > term[v]$
28: $state[i] \leftarrow Primary$
29: $state[j] \leftarrow Secondary, \forall j \in (Q \setminus \{i\})$
30: $term[j] \leftarrow term[i] + 1, \forall j \in Q$
31: $configTerm[i] \leftarrow term[i] + 1$
32:
33: **action:** UPDATETERMS(i, j)
34: **require** $term[i] > term[j]$
35: $state[j] \leftarrow Secondary$
36: $term[j] \leftarrow term[i]$

have some mechanism, however, for determining whether one configuration is newer than another. This is achieved by totally ordering configurations by their $(version, term)$ pair, where term is compared first, followed by version. If configuration C_j compares as greater than configuration C_i based on this ordering, we say that C_j is *newer* than C_i. A secondary can update its configuration to any that is newer than its current configuration. If it learns that another server has a newer configuration, it will fetch that server's configuration, verify that it is still newer than its own upon receipt, and install it locally.

The above provides a basic outline of how reconfigurations occur and how configurations are propagated between servers in *MongoRaftReconfig*. The pseudocode given in Algorithm 1 gives a more abstract and precise description of these behaviors. Note that, in order for the protocol to operate safely, there are several additional restrictions that are imposed on both reconfigurations and elections, which we discuss in more detail below, in Sections 3.3 and 3.4.

3.3 Safety Restrictions on Reconfigurations

In *MongoStaticRaft*, which does not allow reconfiguration, the safety of the protocol depends on the fact that the *quorum overlap* condition is satisfied for the member sets of any two configurations. This holds since there is a single, uniform configuration that is never modified. For any pair of arbitrary configurations, however, their member sets may not satisfy this property. So, in order for *MongoRaftReconfig* to operate safely, extra restrictions are needed on how nodes are allowed to move between configurations. First, any reconfiguration that moves from C to C' is required to satisfy the quorum overlap condition i.e. $QuorumsOverlap(C.m, C'.m)$. This restriction is discussed in Raft's approach to reconfiguration [19], and is adopted by *MongoRaftReconfig*. Even if quorum overlap is ensured

between any two adjacent configurations, it may not be ensured between all configurations that the system passes through over time. So, there are additional preconditions that must be satisfied before a primary server in term T can execute a reconfiguration out of its current configuration C:

$Q1$. *Config Quorum Check*: There must be a quorum of servers in $C.m$ that are currently in configuration C.

$Q2$. *Term Quorum Check*: There must be a quorum of servers in $C.m$ that are currently in term T.

$P1$. *Oplog Commitment*: All oplog entries committed in terms $\leq T$ must be committed on some quorum of servers in $C.m$.

The above preconditions are stated in Algorithm 1 as $Q1(i)$, $Q2(i)$, and $P1(i)$, and they collectively enforce two fundamental requirements needed for safe reconfiguration: *deactivation* of old configurations and *state transfer* from old configurations to new configurations. Q1, when coupled with the election restrictions discussed in Section 3.4, achieves deactivation by ensuring that configurations earlier than C can no longer elect a primary. Q2 ensures that term information from older configurations is correctly propagated to newer configurations, while P1 ensures that previously committed oplog entries are properly transferred to the current configuration, ensuring that any primary in a current or later configuration will contain these entries.

3.4 Configurations and Elections

When a node runs for election in *MongoStaticRaft*, it must ensure its log is appropriately up to date and that it can garner a quorum of votes in its term. In *MongoRaftReconfig*, there is an additional restriction on voting behavior that depends on configuration ordering. If a replica set server is a candidate for election in configuration C_i, then a prospective voter in configuration C_j may only cast a vote for the candidate if C_i is newer than or equal to C_j. Furthermore, when a node wins an election, it must update its current configuration with its new term before it is allowed to execute subsequent reconfigurations. That is, if a node with current configuration (m, v, t) wins election in term t', it will update its configuration to (m, v, t') before allowing any reconfigurations to be processed. This behavior is necessary to appropriately deactivate concurrent reconfigurations that may occur on primaries in a different term. This configuration re-writing behavior is analogous to the write in Raft's corrected membership change protocol proposed in [20].

3.5 Formal Specification

The complete, formal description of *MongoRaftReconfig* is given in the TLA+ specification in the supplementary materials [26]. Note that TLA+ does not impose an underlying system or communication model (e.g. message passing, shared memory), which allows one to write specifications at a wide range of abstraction levels. Our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol and system model. In practice, we have found the abstraction level of our specifications most useful for understanding and communicating the essential behaviors and safety characteristics of the protocol, while also serving to make automated verification via model checking more feasible, which is discussed further in Section 4.4.

4 Correctness

In this section we present a brief outline of our safety proof for *MongoRaftReconfig*. We do not address liveness properties in this work. The full proof is left to [28].

The key, high level safety property of *MongoRaftReconfig* that we establish in this paper is *LeaderCompleteness*, which is a fundamental safety property of both standard Raft and *MongoStaticRaft*, and is stated below as Theorem 2. This property states that if a log entry has been committed in term T, then it must be present in the logs of all primary servers in terms $> T$. Essentially, it ensures that writes committed by some primary will be permanently durable in the replica set. Below we give a high level, intuitive outline of the proof.

4.1 Overview

Conceptually, *MongoRaftReconfig* can be viewed as an extension of the *MongoStaticRaft* replication protocol that allows for dynamic reconfiguration. *MongoRaftReconfig*, however, violates the property that all quorums of any two configurations overlap, which *MongoStaticRaft* relies on for safety. It is therefore necessary to examine how *MongoRaftReconfig* operates safely even though it cannot rely on the quorum overlap property. In *MongoStaticRaft*, there are two key aspects of protocol behavior that depend on quorum overlap: (1) *elections* of primary servers and (2) *commitment* of log entries. Elections must ensure that there is at most one unique primary per term, referred to as the *ElectionSafety* property. Additionally, if a log entry is committed in a given term, it must be present in the logs of all primary servers in higher terms, referred to as the *LeaderCompleteness* property. Both of these safety properties must be upheld in *MongoRaftReconfig*.

LeaderCompleteness is the essential, high level safety property that we must establish for *MongoRaftReconfig*. *ElectionSafety* is a key, auxiliary lemma that is required in order to show *LeaderCompleteness*. So, this guides the general structure of our proof. Section 4.2 presents an intuitive outline of the *ElectionSafety* proof, followed by a similar discussion of *LeaderCompleteness* in Section 4.3. The full proofs are left to [28].

4.2 Election Safety

In *MongoStaticRaft*, if an election has occurred in term T it ensures that some quorum of servers have terms $\geq T$. This prevents any future candidate from being elected in term T, since the quorum required for any future election will contain at least one of these servers, preventing a successful election in term T. This property, referred to as *ElectionSafety*, is stated below as Lemma 1.

▶ **Lemma 1** (Election Safety). *For all $s, t \in Server$ such that $s \neq t$, it is not the case that both s and t are primary and have the same term.*

$$\forall s, t \in Server :$$
$$(state[s] = Primary \wedge state[t] = Primary \wedge term[s] = term[t]) \Rightarrow (s = t)$$

In *MongoRaftReconfig*, ensuring that a quorum of nodes have terms $\geq T$ after an election in term T is not sufficient to ensure that *ElectionSafety* holds, since there is no guarantee that all quorums of future configurations will overlap with those of past configurations. To address this, *MongoRaftReconfig* must appropriately *deactivate* past configurations before creating new configurations. Conceptually, configurations in the protocol can be considered as either or

active or *deactivated*, the former being any configuration that is not deactivated. Deactivated configurations cannot elect a new leader or execute a reconfiguration. *MongoRaftReconfig* ensures proper deactivation of configurations by upholding an invariant that the quorums of all active configurations overlap with each other. In addition to deactivation of configurations, *MongoRaftReconfig* must also ensure that term information from one configuration is properly transferred to subsequent configurations, so that later configurations know about elections that occurred in earlier configurations. For example, if an election occurred in term T in configuration C, even if C is deactivated by the time C' is created, the protocol must also ensure that C' is "aware" of the fact that an election in T occurred in C. *MongoRaftReconfig* ensures this by upholding an additional invariant stating that the quorums of all active configurations overlap with some server in term $\geq T$, for any past election that occurred in term T.

Collectively, the two above invariants are the essential properties for understanding how the *ElectionSafety* property is upheld in *MongoRaftReconfig*. The formal statement of these invariants and the complete proof is left to [28]. In the following section, we briefly discuss the *LeaderCompleteness* property and its proof, which relies on the *ElectionSafety* property.

4.3 Leader Completeness

LeaderCompleteness is the key high level safety property of *MongoRaftReconfig*. It ensures that if a log entry is committed in term T, then it is present in the logs of all leaders in terms $> T$. Essentially, it ensures that committed log entries are durable in a replica set. It is stated below as Theorem 2, where $committed \in \mathbb{N} \times \mathbb{N}$ refers to the set of committed log entries as $(index, term)$ pairs, and $InLog(i, t, s)$ is a predicate determining whether a log entry (i, t) is contained in the log of server s.

▶ **Theorem 2** (Leader Completeness). *If a log entry is committed in term T, then it is present in the log of any leader in term $T' > T$.*

$$\forall s \in Server : \forall (cindex, cterm) \in committed : \\ (state[s] = Primary \wedge cterm < term[s]) \Rightarrow InLog(cindex, cterm, s) \tag{3}$$

In *MongoStaticRaft*, *LeaderCompleteness* is ensured due to the overlap between quorums used for commitment of a write and quorums used for the election of a primary. In *MongoRaftReconfig*, this does not hold, so the protocol instead upholds a more general invariant, stating that, for all committed entries E, the quorums of all active configurations overlap with some server that contains E in its log. *MongoRaftReconfig* also ensures that newer configurations appropriately disable commitment of log entries in older terms. We defer the statement of these invariants and the complete proof of Theorem 2 and its supporting lemmas to [28].

4.4 Model Checking

In addition to the safety proof outlined above, we used TLC [33], an explicit state model checker for TLA+ specifications, to gain additional confidence in the safety of the protocol. We consider it important to augment the human reasoning process for protocols like this with some type of machine based verification, even if the verification is incomplete, since it is easy for humans to make subtle errors in reasoning when considering distributed protocols of this nature.

We verified fixed, finite instances of *MongoRaftReconfig* to provide a sound guarantee of protocol correctness for given parameters. *MongoRaftReconfig* is an infinite state protocol, so verification via explicit state model checking is, necessarily, incomplete. That is, it does not establish correctness of the protocol for an unbounded number of servers or system parameters. It does, however, provide a strong initial level of confidence that the protocol is safe. A goal for future work is to develop a complete, machine checked safety proof using the TLA+ proof system [5].

4.4.1 Methodology and Results

Formally, *MongoRaftReconfig* behaves as an extension of *MongoStaticRaft* that allows for dynamic reconfiguration. Thus, it can be viewed as a composition of two distinct subprotocols: one for managing the oplog, and one for managing configuration state. The oplog is maintained by *MongoStaticRaft*, and configurations are maintained by a protocol we refer to below as *MongoLoglessDynamicRaft*, which implements the logless replicated state machine that manages the configuration state of the replica set. Algorithm 1 summarizes the behaviors of *MongoLoglessDynamicRaft*. This compositional approach to describing *MongoRaftReconfig* is formalized in our TLA+ specification which can be found in the supplementary materials [26]. Our verification efforts centered on checking the two key safety properties discussed in the above sections, *ElectionSafety* and *LeaderCompleteness*. We summarize the results below, leaving the full details to [28].

Checking Leader Completeness. We were able to successfully verify the *LeaderCompleteness* property on a finite instance of *MongoRaftReconfig* with 4 servers, logs of maximum length 2, maximum configuration versions of 3, and maximum server terms of 3. That is, we manually imposed a constraint preventing the model checker from exploring any states exceeding these finite bounds. Model checking this instance generated approximately 345 million distinct protocol states and took approximately 8 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU.

Checking Election Safety. As evidenced by the above metrics, it was difficult to scale verification of the *LeaderCompleteness* property to much larger system parameters. So, to provide additional confidence, we checked the *ElectionSafety* property on the *MongoLoglessDynamicRaft* protocol in isolation, which allowed us to verify instances with significantly larger parameters. Due to the compositional structure of *MongoRaftReconfig*, verifying that the *ElectionSafety* property holds on *MongoLoglessDynamicRaft* is sufficient to ensure that it holds in *MongoRaftReconfig*. Intuitively, the additional preconditions imposed by *MongoRaftReconfig* only *restrict* the behaviors of *MongoLoglessDynamicRaft*, but do not augment them. We formalize and prove this fact via a refinement based argument, whose details are left to [28]. This allows us to assume our verification efforts for *MongoLoglessDynamicRaft* hold in *MongoRaftReconfig*, providing stronger confidence in the correctness of the overall protocol.

We successfully verified the *ElectionSafety* property on a finite instance of *MongoLoglessDynamicRaft* with 5 servers, maximum configuration versions of 4, and maximum terms of 4. Model checking this instance generated approximately 812 million distinct states and took around 19.5 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU. The ability to check these considerably larger parameter values in only several extra hours of wall clock time demonstrates the effectiveness of this compositional model checking approach, helping us mitigate state space explosion [6].

5 Conceptual Insights

MongoRaftReconfig can be viewed as a generalization and optimization of the standard Raft reconfiguration protocol. To explain the conceptual novelties of our protocol and how it relates to standard Raft, we discuss below the two primary aspects of the protocol which set it apart from Raft: (1) decoupling of the oplog and config state machine and (2) logless optimization of the config state machine. These are covered in Sections 5.1 and 5.2, respectively. Section 6 provides an experimental evaluation of how these novel aspects can provide performance benefits for reconfiguration, by allowing reconfigurations to bypass the main operation log in cases where it has become slow or stalled.

5.1 Decoupling Reconfigurations

In standard Raft, the main operation log is used for both normal operations and reconfiguration operations. This coupling between logs has the benefit of providing a single, unified data structure to manage system state, but it also imposes fundamental restrictions on the operation of the two logs. Most importantly, in order for a write to commit in one log, it must commit all previous writes in the other. For example, if a reconfiguration log entry C_j has been written at log index j on primary s, and there is a sequence of uncommitted log entries $U = \langle i, i+1, ..., j-1 \rangle$ in the log of s, in order for a reconfiguration from C_j to C_k to occur, all entries of U must become committed. This behavior, however, is stronger than necessary for safety i.e. it is not strictly necessary to commit these log entries before executing a reconfiguration. The only fundamental requirements are that previously committed log entries are committed by the rules of the current configuration, and that the current configuration has satisfied the necessary safety preconditions. Raft achieves this goal implicitly, but more conservatively than necessary, by committing the entry C_j and all entries behind it. This ensures that all previously committed log entries, in addition to the uncommitted operations U, are now committed in C_j, but it is not strictly necessary to pipeline a reconfiguration behind commitment of U. *MongoRaftReconfig* avoids this by separating the oplog and config state machine and their rules for commitment and reconfiguration, allowing reconfigurations to bypass the oplog if necessary. Section 6 examines this aspect of the protocol experimentally.

5.2 Logless Optimization

Decoupling the config state machine from the main operation log allows for an optimization that is enabled by the fact that reconfigurations are "update-only" operations on the replicated state machine. This means that it is sufficient to store only the latest version of the replicated state, since the latest version can be viewed as a "rolled-up" version of the entire (infinite) log. This logless optimization allows the configuration state machine to avoid complexities related to garbage collection of old log entries and it simplifies the mechanism for state propagation between servers. Normally, log entries are replicated incrementally, either one at a time, or in batches from one server to another. Additionally, servers may need to have an explicit procedure for deleting (i.e. rolling back) log entries that will never become committed. In the logless replicated state machine, all of these mechanisms can be combined into a single conceptual action, that atomically transfers the entire log of server s to another server t, if the log of s is newer, based on the index and term of its last entry. In *MongoRaftReconfig*, this is implemented by the *SendConfig* action, which transfers configuration state from one server to another.

6 Experimental Evaluation

In a healthy replica set, it is possible that a failure event causes some subset of replica set servers to degrade in performance, causing the main oplog replication channel to become lagged or stall entirely. If this occurs on a majority of nodes, then the replica set will be prevented from committing new writes until the performance degradation is resolved. For example, consider a 3 node replica set consisting of nodes $\{n0, n1, n2\}$, where nodes $n1$ and $n2$ suddenly become slow or stall replication. An operator or failure detection module may want to reconfigure these nodes out of the set and add in two new, healthy nodes, $n3$ and $n4$, so that the system can return to a healthy operational state. This requires a series of two reconfigurations, one to add $n3$ and one to add $n4$. In standard Raft, this would require the ability to commit at least one reconfiguration oplog entry with one of the degraded nodes ($n1$ or $n2$). This prevents such a reconfiguration until the degradation is resolved. In *MongoRaftReconfig*, reconfigurations bypass the oplog replication channel, committing without the need to commit writes in the oplog. This allows *MongoRaftReconfig* to successfully reconfigure the system in such a degraded state, restoring oplog write availability by removing the failed nodes and adding in new, healthy nodes.

Note that if a replica set server experiences a period of degradation (e.g. a slow disk), both the oplog and reconfiguration channels will be affected, which would seem to nullify the benefits of decoupling the reconfiguration and oplog replication channels. In practice, however, the operations handled by the oplog are likely orders of magnitude more resource intensive than reconfigurations, which typically involve writing a negligible amount of data. So, even on a degraded server, reconfigurations should be able to complete successfully when more intensive oplog operations become prohibitively slow, since the resource requirements of reconfigurations are extremely lightweight.

6.1 Experiment Setup and Operation

To demonstrate the benefits of *MongoRaftReconfig* in this type of scenario, we designed an experiment to measure how quickly a replica set can reconfigure in new nodes to restore majority write availability when it faces periodic phases of degradation. For comparison, we implemented a simulated version of the Raft reconfiguration algorithm in MongoDB by having reconfigurations write a no-op oplog entry and requiring it to become committed before the reconfiguration can complete [25]. Our experiment initiates a 5 node replica set with servers we refer to as $\{n0, n1, n2, n3, n4\}$. We run the server processes co-located on a single Amazon EC2 t2.xlarge instance with 4 vCPU cores, 16GB memory, and a 100GB EBS disk volume, running Ubuntu 20.04. Co-location of the server processes is acceptable since the workload of the experiment does not saturate any resource (e.g. CPU, disk) of the machine. The servers run MongoDB version v4.4-39f10d with a patch to fix a minor bug [18] that prevents optimal configuration propagation speed in some cases.

Initially, $\{n0, n1, n2\}$ are voting servers and $\{n3, n4\}$ are non voting. In a MongoDB replica set, a server can be assigned either 0 or 1 votes. A non-voting server has zero votes and it does not contribute to a commit majority i.e. it is not considered as a member of the consensus group. Our experiment has a single writer thread that continuously inserts small documents into a collection with write concern *majority*, with a write concern timeout of 100 milliseconds. There is a concurrent fault injector thread that periodically simulates a degradation of performance on two secondary nodes by temporarily pausing oplog replication on those nodes. This thread alternates between *steady* periods and *degraded* periods of time, starting out in *steady* mode, where all nodes are operating normally. It runs for 5 seconds in

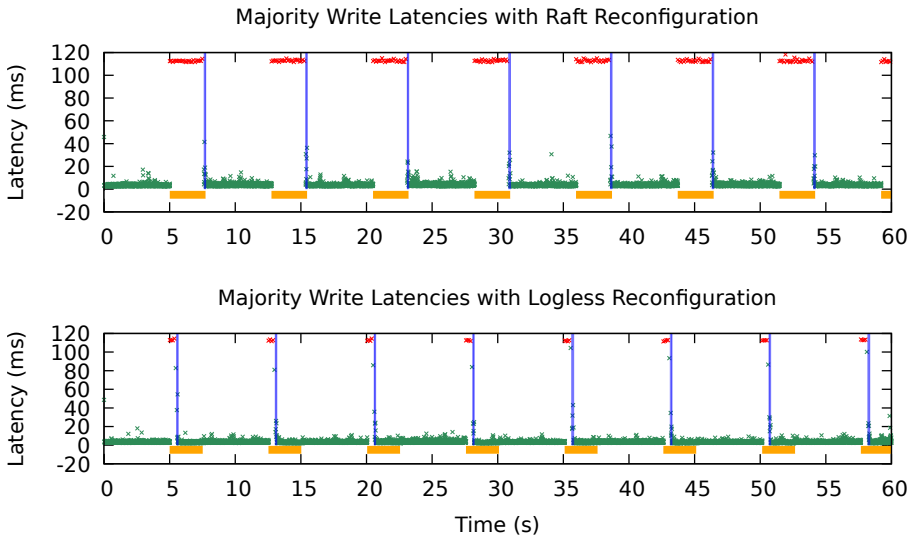

Figure 1 Latency of majority writes in the face of node degradation and reconfiguration to recover. Red points indicate writes that timed out i.e. failed to commit. Orange horizontal bars indicate intervals of time where system entered a *degraded* mode. Thin, vertical blue bars indicate successful completion of reconfiguration events.

steady mode, then transitions to *degraded* mode for 2.5 seconds, before transitioning back to *steady* mode and repeating this cycle. When the fault injector enters *degraded* mode, the main test thread simulates a "fault detection" scenario (assuming some external module detected the performance degradation) by sleeping for 500 milliseconds, and then starting a series of reconfigurations to add two new, healthy secondaries and remove the two degraded secondaries. Over the course of the experiment, which has a 1 minute duration, we measure the latency of each operation executed by the writer thread. These latencies are depicted in the graphs of Figure 1. Red points indicate writes that failed to commit i.e. that timed out at 100 milliseconds. The successful completion of reconfigurations are depicted with vertical blue bars. It can be seen how, when a period of degradation begins, the logless reconfiguration protocol is able to complete a series of reconfigurations quickly to get the system back to a healthy state, where writes are able to commit again and latencies drop back to their normal levels. In the case of Raft reconfiguration, writes continue failing until the period of degradation ends, since the reconfigurations to add in new healthy nodes cannot complete.

7 Related Work

Dynamic reconfiguration in consensus based systems has been explored from a variety of perspectives for Paxos based systems. In Lamport's presentation of Paxos [13], he suggests using a fixed parameter α such that the configuration for a consensus instance i is governed by the configuration at instance $i - \alpha$. This restricts the number of commands that can be executed until the new configuration becomes committed, since the system cannot execute instance i until it knows what configuration to use, potentially causing availability issues if reconfigurations are slow to commit. Stoppable Paxos [14] was an alternative method later proposed where a Paxos system can be reconfigured by stopping the current state machine

and starting up a new instance of the state machine with a potentially different configuration. This "stop-the-world" approach can hurt availability of the system while a reconfiguration is being processed. Vertical Paxos allows a Paxos state machine to be reconfigured in the middle of reaching agreement, but it assumes the existence of an external configuration master [15]. In [4], the authors describe the Paxos implementation underlying Google's Chubby lock service, but do not include details of their approach to dynamic reconfiguration, stating that "While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos...". They remark that the details, though minor, are "...subtle and beyond the scope of this paper".

The Raft consensus protocol, published in 2014 by Ongaro and Ousterhout [22], presented two methods for dynamic membership changes: single server membership change and joint consensus. A correctness proof of the core Raft protocol, excluding dynamic reconfiguration, was included in Ongaro's PhD dissertation [19]. Formal verification of Raft's linearizability guarantees was later completed in Verdi [32], a framework for verifying distributed systems in the Coq proof assistant [3], but formalization of dynamic reconfiguration was not included. In 2015, after Raft's initial publication, a safety bug in the single server reconfiguration approach was found by Amos and Zhang [2], at the time PhD students working on a project to formalize parts of Raft's original reconfiguration algorithm. A fix was proposed shortly after by Ongaro [20], but the project was never extended to include the fixed version of the protocol. The Zab replication protocol, implemented in Apache Zookeeper [29], also includes a dynamic reconfiguration approach for primary-backup clusters that is similar in nature to Raft's joint consensus approach.

The concept of decoupling reconfiguration from the main data replication channel has previously appeared in other replication systems, but none that integrate with a Raft-based system. RAMBO [8], an algorithm for implementing a distributed shared memory service, implements a dynamic reconfiguration module that is loosely coupled with the main read-write functionality. Additionally, Matchmaker Paxos [31] is a more recent approach for reconfiguration in Paxos based protocols that adds dedicated nodes for managing reconfigurations, which decouples reconfiguration from the main processing path, preventing performance degradation during configuration changes. There has also been prior work on reconfiguration using weaker models than consensus [10], and approaches to logless implementations of Paxos based replicated state machine protocols [23], which bear conceptual similarities to our logless protocol for managing configuration state. Similarly, [11] presents an approach to asynchronous reconfiguration under a Byzantine fault model that avoids reaching consensus on configurations by utilizing a lattice agreement abstraction.

8 Conclusions and Future Work

In this paper we presented *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol that improves upon and generalizes the single server reconfiguration protocol of standard Raft by decoupling the main operation and reconfiguration logs. Although *MongoRaftReconfig* was developed for and presented in the context of the MongoDB system, the ideas and underlying protocol generalize to other Raft-based replication protocols that require dynamic reconfiguration. Goals for future work include development of a machine checked safety proof of the protocol's correctness with help of the TLA+ proof system [5], in addition to running more in depth experiments to evaluate how *MongoRaftReconfig* behaves under more varied workloads.

References

1 Marcos Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 2010.

2 Brandon Amos and Huanchen Zhang. Specifying and proving cluster membership for the Raft distributed consensus algorithm, 2015. URL: `https://www.cs.cmu.edu/~aplatzer/course/pls15/projects/bamos.pdf`.

3 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

4 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1281100.1281103`.

5 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.

6 Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

7 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, 2012. `doi:10.1145/2518037.2491245`.

8 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 2010. `doi:10.1007/s00446-010-0117-1`.

9 Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 2020. `doi:10.14778/3415478.3415535`.

10 Leander Jehl and Hein Meling. Asynchronous reconfiguration for Paxos state machines. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. `doi:10.1007/978-3-642-45249-9_8`.

11 Petr Kuznetsov and Andrei Tonkikh. Asynchronous Reconfiguration with Byzantine Failures. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.DISC.2020.27`.

12 Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998. `doi:10.1145/279227.279229`.

13 Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 2001. `doi:10.1145/568425.568433`.

14 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

15 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1582716.1582783`.

16 Stephan Merz. *The Specification Language TLA+*, pages 401–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-74107-7_8.

17 MongoDB Github Project, 2021. URL: https://github.com/mongodb/mongo.

18 MongoDB JIRA Ticket SERVER-46907, 2020. URL: https://jira.mongodb.org/browse/SERVER-46907.

19 Diego Ongaro. Consensus: Bridging Theory and Practice. *Doctoral thesis*, 2014.

20 Diego Ongaro. Bug in single-server membership changes, July 2015. URL: https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J.

21 Diego Ongaro. The Raft Consensus Algorithm, 2021. URL: https://raft.github.io/.

22 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association.

23 Denis Rystsov. CASPaxos: Replicated State Machines without logs, 2018. arXiv:1802.07000.

24 Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990. doi:10.1145/98163.98167.

25 William Schultz. MongoDB Experiment Source Code, September 2021. URL: https://github.com/will62794/mongo/tree/2bb9f30da.

26 William Schultz. MongoRaftReconfig TLA+ Specifications, November 2021. doi:10.5281/zenodo.5715511.

27 William Schultz, Tess Avitabile, and Alyson Cabral. Tunable Consistency in MongoDB. *Proc. VLDB Endow.*, 12(12):2071–2081, August 2019. doi:10.14778/3352063.3352125.

28 William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. *arXiv preprint*, 2021. arXiv:2102.11960.

29 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012*, 2019.

30 Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3386134.

31 Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A Reconfigurable Consensus Protocol [Technical Report], 2020. arXiv:2007.09468.

32 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016 - Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, co-located with POPL 2016*, 2016. doi:10.1145/2854065.2854081.

33 Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

34 Siyuan Zhou and Shuai Mu. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *NSDI*, pages 687–703, 2021.