Design and Verification of a Logless Dynamic Reconfiguration Protocol in MongoDB Replication

WILLIAM SCHULTZ, Northeastern University, United States
SIYUAN ZHOU, MongoDB, Inc., United States
STAVROS TRIPAKIS, Northeastern University, United States

We present a novel dynamic reconfiguration protocol for the MongoDB replication system that extends and generalizes the single server reconfiguration protocol of the Raft consensus algorithm. Our protocol decouples the processing of configuration changes from the main database operation log, which allows reconfigurations to proceed in cases when the main log is prevented from processing new operations. Additionally, this decoupling allows for configuration state to be managed by a *logless* replicated state machine, by optimizing away the explicit log and storing only the latest version of the configuration, avoiding the complexities of a log-based protocol. We provide a formal specification of the protocol along with results from automated verification of its safety properties. We also provide an experimental evaluation of the protocol benefits, showing how reconfigurations are able to quickly restore a system to healthy operation in scenarios where node failures have stalled the main operation log.

1 INTRODUCTION

Distributed replication systems based on the replicated state machine model [35] 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 [10, 12, 14, 43, 47]. Paxos [16] and, more recently, the Raft algorithm [31], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [5] 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 [25] or require the use of an external configuration master [20]. 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 [29] in one of its reconfiguration protocols was found after initial publication, which has since been fixed. The discovery of bugs like these demonstrate that the design and verification of a safe dynamic reconfiguration protocol is a non-trivial task, and the correctness of published protocols may not be particularly well understood by system designers and engineers, which is often important if optimizations or modifications need to be made to an underlying protocol. As a rule of thumb, we believe that if distributed systems researchers can make errors designing these protocols, it is even harder for system engineers to understand the subtleties of these protocols when implementing them. Thus, it is important for their behaviors and characteristics to be well studied, formalized, and understood in different contexts and systems.

MongoDB [3] is a general purpose, document-oriented database which implements a distributed replication system [41] for providing high availability and fault tolerance. Since its inception, MongoDB's distributed replication system provided a mechanism for clients to dynamically reconfigure replica membership, but this legacy protocol was unsafe in certain cases. In recent versions of MongoDB, reconfiguration has become a more common operation which necessitated the need for a redesigned, safe reconfiguration protocol. The new reconfiguration protocol was designed with a goal of provable correctness guarantees and minimizing changes to the existing protocol where possible. In this paper we propose a new protocol that satisfies these goals, and that improves upon and generalizes the single server reconfiguration protocol of standard Raft. This protocol, which we refer to as *MongoRaftReconfig*, provides *logless* dynamic reconfiguration by decoupling the processing of configuration changes from the main database operation log. This allows for design modularity in addition to improving reconfiguration performance by letting reconfigurations run in parallel to the main operation log. We present this protocol along with a formal specification and verification of its safety properties and an experimental evaluation of its performance benefits. We additionally provide a discussion of how our protocol relates to and generalizes Raft's original reconfiguration protocol, which helps to establish a deeper understanding of Raft's behaviors and properties.

To summarize, we make the following contributions:

- We present *MongoRaftReconfig*, a novel extension of the static MongoDB replication protocol that allows for logless dynamic reconfiguration, and which generalizes and optimizes the single server reconfiguration protocol of standard Raft.
- We provide a formal specification of *MongoRaftReconfig* in TLA+ [26] along with results from automated verification of its key safety properties using the TLC model checker [46].
- We discuss how the concepts and behaviors of our protocol can be mapped to reconfiguration in standard Raft. Specifically, we show our how protocol optimizes Raft reconfiguration by avoiding unnecessary commitment of writes during reconfigurations, and how it simplifies the log structure for managing configuration state.
- We provide an experimental evaluation of *MongoRaftReconfig*'s 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 a *member set* as an element $m \in \mathcal{P}(Server)$, where $\mathcal{P}$ is the powerset operator. We define a *quorum* similarly, as an element $q \in \mathcal{P}(Server)$. Member sets and quorums have the same type but refer to different conceptual entities. For any member set m, there is an associated set of quorums, denoted $Quorums(m)$, which contains all quorums in $\mathcal{P}(m)$ with at least a majority of elements in m.

$$Quorums(m) = \{s \in \mathcal{P}(m) : |s| * 2 > |m|\} \tag{1}$$

where $|S|$ denotes the cardinality of a set S. For two member sets m_i, m_j, we say that they satisfy the *quorum overlap* condition if any two quorums of either set have at least one common member i.e.

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

For any two member sets m_i, m_j that differ by at most one element, the quorum overlap condition is satisfied i.e.

$$\forall m_i, m_j \in \mathcal{P}(Server) : |m_i \Delta m_j| \leq 1 \Rightarrow QuorumOverlap(m_i, m_j) \tag{3}$$

where Δ represents the *symmetric difference* between two sets. This fact is demonstrated in Section 4.1 of [28].

2.2 Raft

Raft [28] 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 [30]. 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 different, alternate algorithms: *single server membership change* and *joint consensus*. Joint consensus adopts a two phase approach, where the system must move through an intermediate configuration before reaching a specified target configuration. The single server change approach avoids this complexity by restricting reconfigurations to only add or remove a single node. In both algorithms, reconfiguration is accomplished by writing a special reconfiguration entry into the main Raft operation log that alters the local configuration of a node. Throughout, we refer to the original Raft protocol as described and specified in [28] as *standard Raft*. In this paper we are primarily concerned with Raft's single server change reconfiguration protocol, so 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 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 node maintains a logical copy of the database state.

2.3.1 Static Replication Protocol. MongoDB replica sets utilize a replication protocol that is similar to 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 [41, 48], but we provide a high level overview here, since our reconfiguration protocol, *MongoRaftReconfig*, 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 node's 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*, meaning 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.

3 *MongoRaftReconfig*: A LOGLESS DYNAMIC RECONFIGURATION PROTOCOL

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 [25, 31, 42] 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 replicated state machine, which we refer to as the *config log*. The config log is maintained alongside the oplog, and manages the configuration state used by the overall protocol.

Decoupling these two conceptually distinct logs, the oplog and the config log, enables certain optimizations and simplifications in *MongoRaftReconfig* which would not be possible in a protocol where both logs are interleaved with each other. First, it allows for a simplification of the config log structure by observing that configuration changes are an "update only" operation. This obviates the need to store the entire log history, allowing the config log to operate as a *logless* replicated state machine, storing only the latest version of the configuration state. Second, it prevents the dynamics of either log negatively impacting the other unnecessarily. For example, it is possible to commit writes in either log independently, without requiring previous writes in the other log to also become committed. This can allow the config log to bypass the oplog, allowing for reconfigurations in cases where a slow or stalled oplog replication channel would otherwise prevent reconfigurations from proceeding. We examine these benefits experimentally in Section 6.2. In the remainder of this section we give a high level overview of the behaviors of *MongoRaftReconfig* and how it operates safely. In Section 4 we present our formal specification of the protocol in TLA+, which allows for a more precise description and enables automated verification of the protocol's safety properties, which we present in Section 5.

3.1 Protocol Behavior

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 3.1.2. Formally, a *configuration* is defined as a tuple (m, v, t), where m is a non-empty member set, $v \in \mathbb{N}$ is a numeric configuration *version*, and $t \in \mathbb{N}$ is the numeric *term* of the configuration[1]. Each server of a replica set maintains a single, durable configuration, and it is assumed that, initially, all nodes begin with a common configuration.

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_{new}. Reconfigurations can only be executed on primary servers, and they update the primary's current local configuration C_{old} to the specified configuration C_{new}. The version of the new configuration, $C_{new}.v$, must be greater than the version of the primary's current configuration, $C_{old}.v$, and the term of C_{new} 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 have some mechanism, however, for determining whether another configuration is newer than their own. 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 install any configuration that is newer than its own. If it learns that some other 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.

[1]For sake of convenience, we refer to the elements of a configuration tuple $C = (m, v, t)$ as, respectively, $C.m$, $C.v$ and $C.t$.

The above provides a basic outline of how reconfigurations occur and how configurations are propagated between servers in *MongoRaftReconfig*. In order for the protocol to operate safely, however, there are several additional restrictions that are imposed on both reconfigurations and elections, which we discuss in more detail below.

3.1.1 Safety Restrictions on Reconfigurations. In *MongoStaticRaft*, which does not allow reconfiguration, the safety of the protocol depends on the fact that *quorum overlap* is satisfied for the member sets of any two configurations, 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_{old} to C_{new} is required to satisfy the quorum overlap condition i.e. $QuorumOverlap(C_{old}.m, C_{new}.m)$. To satisfy this, it is sufficient to enforce a *single node change* condition, which requires that no more than a single member is added or removed in a single reconfiguration. The sufficiency of this condition to enforce quorum overlap is illustrated in Formula 3. Although the single node change condition ensures quorum overlap between two adjacent configurations, it may not be ensured between all configurations that the system passes through over time. So, there are two additional preconditions that must be satisfied before a primary node can execute a reconfiguration out of its current configuration C.

P1. *Config Commitment*: The primary's current configuration, C, must be replicated to and installed on some quorum of servers in its current member set, $C.m$, that are in the primary's current term.
P2. *Oplog Commitment*: Any oplog entries that were committed by the current primary in its previous configuration must be committed on some quorum of servers in its current member set, $C.m$.

At a high level, these preconditions enforce, respectively, two fundamental requirements needed for safe reconfiguration: *deactivation* of old configurations and *state transfer* from old configurations to new configurations. P1 ensures that configurations earlier than C can no longer independently form a quorum for electing a node or committing a log entry. P2 ensures that previously committed oplog entries are properly transferred to the current configuration, which ensures that any primary elected in a subsequent configuration will contain these entries.

3.1.2 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 voters 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 commit a configuration in its own term before it is allowed to execute subsequent reconfigurations. This is achieved by requiring nodes to atomically re-write their existing configuration with their new term upon winning election in term t. 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 disable 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 [29].

3.2 Summary

The above provides a high level description of the behaviors of *MongoRaftReconfig* and how it operates safely. In the following section we present our formal specification of the protocol in TLA+, which allows us to define the protocol and its safety properties precisely. Additionally, it allows for automated verification of the protocol's correctness, which we discuss in Section 5.

4 FORMAL SPECIFICATION

MongoRaftReconfig behaves as an extension of *MongoStaticRaft* that allows for dynamic reconfiguration. Thus, it can be formally viewed as a composition of two distinct subprotocols: one for managing the oplog, and one for managing the config log. The oplog is maintained by *MongoStaticRaft*, and the config log is maintained by a protocol we refer to as *MongoLoglessDynamicRaft*, which implements a logless replicated state machine that stores the configuration state of the replica set. The complete, formal description of *MongoRaftReconfig* is given in the accompanying TLA+ specification [39], which is summarized in Figure 1. Since *MongoRaftReconfig* is an extension of the existing *MongoStaticRaft* protocol, we first present a specification of *MongoStaticRaft*, followed by the formal specification of *MongoRaftReconfig*.

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 feasible, which we examine further in section 5.2.

4.1 TLA+

We use the TLA+ language [18] to formally describe our reconfiguration protocol. TLA+ is a formal specification language for describing distributed and concurrent systems that is based on first order and temporal logic [33]. Specifying a system in TLA+ consists of defining a set of state variables, $vars$, along with a temporal logic formula which describes the set of permitted system behaviors over these variables. The canonical way of defining a specification is as the conjunction of an initial state predicate, $Init$, and a next state relation, $Next$, which determine, respectively, the set of allowed initial states and how the protocol may transition between states. The overall system is then defined by the temporal formula $Init \wedge \Box[Next]_{vars}$, where $\Box$ denotes the "always" operator of temporal logic, meaning that a formula holds true at every step of a behavior. $[Next]_{vars}$ is equivalent to the expression $Next \vee (vars' = vars)$, which means that specifications of this form allow for *stuttering* steps i.e. transitions that do not change the state. A primed variable, expressed by attaching a ' symbol, denotes the value of a variable in the next state of a system behavior. The next state relation is typically written as a disjunction $A_1 \vee A_2 \vee ... \vee A_n$ of *actions* A_i, where an action is a logical predicate that depends on both the current and next state of a behavior. For example, for variables x and y, the following specification, $Spec$, describes a system whose initial state is $\langle x = 0, y = 0\rangle$, and which at each step can non-deterministically increment x or y by 1, or leave both variables unchanged.

$$Init \triangleq x = 0 \wedge y = 0$$

$$Next \triangleq (x' = x + 1) \vee (y' = y + 1)$$

$$Spec \triangleq Init \wedge \Box[Next]_{\langle x,y\rangle}$$

Correctness properties and system specifications in TLA+ are both written as temporal logic formulas. This allows one to express notions of property satisfaction and refinement in a concise and similar manner. We say that a specification

```
──── MODULE MongoStaticRaft ────
VARIABLE log
VARIABLE committed
VARIABLE currentTerm
VARIABLE state
VARIABLE config

Next ≜
  ∃ s, t ∈ Server :
  ∃ Q ∈ Quorums(config[s]) :
    ∨ ClientRequest(s)
    ∨ GetEntries(s, t)
    ∨ RollbackEntries(s, t)
    ∨ BecomeLeader(s, Q)
    ∨ CommitEntry(s, Q)
    ∨ UpdateTerms(s, t)
────────────────────────────────
```

(a)

```
──── MODULE MongoLoglessDynamicRaft ────
VARIABLE currentTerm
VARIABLE state
VARIABLE config
VARIABLE configVersion
VARIABLE configTerm

Next ≜
  ∃ s, t ∈ Server :
  ∃ Q ∈ Quorums(config[s]) :
    ∨ SendConfig(s, t)
    ∨ Reconfig(s)
    ∨ BecomeLeaderDynamic(s, Q)
    ∨ UpdateTermsDynamic(s, t)
────────────────────────────────────────
```

(b)

```
──── MODULE MongoRaftReconfig ────
VARIABLE log, committed, currentTerm, state, config, configVersion, configTerm

configVars ≜ ⟨configVersion, configTerm⟩
oplogVars ≜ ⟨log, committed⟩

Next ≜
  ∃ s, t ∈ Server :
  ∃ Q ∈ Quorums(config[s]) :
    ∨ ClientRequest(s)          ∧ UNCHANGED configVars
    ∨ GetEntries(s, t)          ∧ UNCHANGED configVars
    ∨ RollbackEntries(s, t)     ∧ UNCHANGED configVars
    ∨ CommitEntry(s, Q)         ∧ UNCHANGED configVars
    ∨ SendConfig(s, t)          ∧ UNCHANGED oplogVars
    ∨ OplogCommitment(s) ∧ Reconfig(s) ∧ UNCHANGED oplogVars
    ∨ BecomeLeader(s, Q) ∧ BecomeLeaderDynamic(s, Q)
    ∨ UpdateTerms(s, t) ∧ UpdateTermsDynamic(s, t)
──────────────────────────────────
```

(c)

Fig. 1. The *MongoRaftReconfig* transition relation defined as a composition of its subprotocols, *MongoStaticRaft* and *MongoLoglessDynamicRaft*. Actions in blue represent those of *MongoStaticRaft*, and actions in orange represent those of *MongoLoglessDynamicRaft*. The UNCHANGED construct indicates that a set of variables do not change on a transition.

S *satisfies* a property P iff the formula $S \Rightarrow P$ is valid (i.e. true under all assignments). We say that a specification S_1 *refines* (or is *a refinement of*) S_2 iff $S_1 \Rightarrow S_2$ is valid i.e. every behavior of S_1 is a valid behavior of S_2 [4].

Notation. TLA+ includes sets, functions, sequences, and records as primitive data types. The expression $Seq(S)$ refers to the set of all sequences with elements in the set S. For a sequence s, the expression $Len(s)$ gives its length, and $s[n]$ refers to the n-th element of s, 1-indexed. Additionally, for a function $f : S \rightarrow T$, we denote $f[s]$ as the value of f on input $s \in S$. For sets S, T, the set of all functions with domain S and range T is denoted by the expression $[S \rightarrow T]$. The construct UNCHANGED x is defined as $x' = x$.

4.2 *MongoStaticRaft* Formal Specification

The high level behaviors of *MongoStaticRaft* were described informally in Section 2.3.1. Here we give a summary of the protocol's formal specification in TLA+, which is provided in [40]. We note that, although the *MongoStaticRaft* protocol existed prior to the work presented in this paper, it did not have a published formal specification. So, we developed one in order to describe how *MongoRaftReconfig* extends the protocol.

The state variables of the *MongoStaticRaft* specification are summarized in Figure 2, and its core actions are shown in Figure 1a. The specification represents the local state of each server in a set of global variables that are functions with domain $Server$. The *log* variable represents the oplog stored on each server, which is a sequence of $term \in \mathbb{N}$ values. The expression $log[s][i]$ refers to the term of the log entry of server s at index i. We alternately refer to such an entry as the $(index, term)$ pair $(i, log[s][i])$. The variable *state* represents whether a server is currently acting in *Primary* or *Secondary* role, and *currentTerm* is the local term of each server. The *committed* variable is a set of $(index, term)$ pairs that represents the set of log entries that have been marked committed. Our specification also includes a *config* variable, which is the member set that each server considers to be part of the replica set. The *config* variable is not strictly required for specifying the behavior of *MongoStaticRaft*, since it is given an initial, uniform value on all servers and never changes. We include it, though, to make it clearer how the protocol is extended to include dynamic reconfiguration, which is described below, in Section 4.3.

In the initial state of the protocol, $state[s] = Secondary$, $currentTerm[s] = 0$, $log[s] = \langle\rangle$, and $config[s] = m$ for all servers $s \in Server$, where $m \in \mathcal{P}(Server)$ is an arbitrary, non-empty member set. The high level descriptions of the core protocol actions are as follows:

- *ClientRequest(s)*: a log entry is written on a primary server s.
- *GetEntries(s,t)*: a log entry is replicated from server s to server t
- *RollbackEntries(s, t)*: server s deletes its last diverged log entry with respect to server t.
- *BecomeLeader(s, Q)*: primary server s is elected with set of voters Q.
- *CommitEntry(s, Q)*: a log entry is marked committed with a set of servers Q by primary server s.
- *UpdateTerms(s, t)*: server t updates its term to the newer term of server s, and reverts to *Secondary* state.

4.3 *MongoRaftReconfig* Formal Specification

We now give an overview of the formal specification of *MongoRaftReconfig*, which extends *MongoStaticRaft* with dynamic reconfiguration. Reconfiguration in *MongoRaftReconfig* is managed by the *MongoLoglessDynamicRaft* subprotocol, so we focus on the behaviors of this subprotocol and how it interacts with *MongoStaticRaft* to form the overall protocol, *MongoRaftReconfig*.

4.3.1 *Variables and Initial States.*

The state variables of *MongoRaftReconfig* along with their corresponding types and the subprotocol in which they are used are summarized in Figure 2. As in *MongoStaticRaft*, the local state of each server is represented in a set of global variables that are functions with domain $Server$. The specification represents the configuration C of a server with three separate state variables, *config*, *configVersion*, and *configTerm*, which represent, respectively, the *member set*, *version*, and *term* of a server's current configuration. That is, the current configuration of a server s is $C = (config[s], configVersion[s], configTerm[s])$. The initial states of the shared protocol variables are the same as in *MongoStaticRaft*, and initially $configVersion[s] = 1$, $configTerm[s] = 0$ for all servers $s \in Server$.

	Protocol	State Variable	Type
MongoRaftReconfig	*MongoStaticRaft*	*log*	$[Server \rightarrow Seq(\mathbb{N})]$
		committed	$\mathcal{P}(\mathbb{N} \times \mathbb{N})$
	Shared	*currentTerm*	$[Server \rightarrow \mathbb{N}]$
		state	$[Server \rightarrow \{Primary, Secondary\}]$
		config	$[Server \rightarrow \mathcal{P}(Server)]$
	MongoLoglessDynamicRaft	*configVersion*	$[Server \rightarrow \mathbb{N}]$
		configTerm	$[Server \rightarrow \mathbb{N}]$

Fig. 2. State variables of the *MongoRaftReconfig* protocol and their corresponding types.

4.3.2 Behaviors. The core actions of *MongoRaftReconfig* are shown in Figure 1c. Reconfigurations are modeled by the *Reconfig(s)* action, shown in Figure 1b of *MongoLoglessDynamicRaft*, which represents a reconfiguration that occurs on server s, and enforces the config commitment precondition, P1. The complete reconfiguration behavior is depicted on line 14 of Figure 1c, which includes the precondition $OplogCommitment(s)$ to enforce condition P2, which depends on the set of committed oplog entries. Configuration propagation is modeled by the *SendConfig(s, t)* action, which represents the propagation of a configuration from server s to server t, and is shown in Figure 1b. The election behavior of *MongoRaftReconfig* is defined as a conjunction of the *BecomeLeader(s, Q)* and *BecomeLeaderDynamic(s, Q)* actions, which represents the election of node s with voter quorum Q and is shown on line 15 of Figure 1c. This conjunction simply means that both actions must be executed jointly i.e. in the same transition. In our specification of *MongoRaftReconfig* we allow term information to propagate between servers at any time. The action *UpdateTerms(s, t)* propagates the term of a server s to server t, where $currentTerm[s] > currentTerm[t]$. Server t updates its term to $currentTerm[s]$ and reverts to *Secondary* state if necessary. The definition of *UpdateTerms(s, t)* and *UpdateTermsDynamic(s,t)* is the same in both *MongoLoglessDynamicRaft* and *MongoStaticRaft*, so the behavior of their composition on line 16 of 1c has the same effect as an *UpdateTerms(s, t)* action.

4.3.3 Protocol Composition. MongoRaftReconfig is specified as a composition of the subprotocols *MongoLoglessDynamicRaft* and *MongoStaticRaft*. The formal definition of this composition is shown in Figure 1c, where actions are colorized according to which subprotocol they belong to. The full definitions of the actions referenced there are given in the complete specification [39].

Specifying *MongoRaftReconfig* as a composition of these two subprotocols helps make the conceptual subcomponents of the protocol clearer, in addition to facilitating the scalability of automated safety verification, which we discuss in more detail in Section 5.2. Importantly, the composition of these protocols does not impact the underlying safety properties of *MongoLoglessDynamicRaft*. That is, *MongoRaftReconfig* only *restricts* the behaviors of *MongoLoglessDynamicRaft*, but does not add any new behaviors. So, any properties that *MongoLoglessDynamicRaft* satisfies in isolation are satisfied when operating as a subprotocol of *MongoRaftReconfig*. We examine this property more formally in Section 5.3.

5 CORRECTNESS ANALYSIS

5.1 Safety Properties

The fundamental safety property of MongoDB's core replication protocol, *MongoStaticRaft*, is the *StateMachineSafety* property, which states that if an oplog entry has been marked committed at a particular log index, no conflicting log entry will ever be marked committed at the same index. We can formally state this property as a predicate on the *committed* variable, which stores the set of committed log entries as $(index, term)$ pairs:

$$StateMachineSafety \triangleq \forall c_i, c_j \in committed : (c_i[1] = c_j[1]) \Rightarrow (c_i = c_j)$$

We want to verify that *MongoRaftReconfig* satisfies the same property. This property is an invariant, meaning that all reachable states of the protocol must satisfy it. Thus, our goal can be stated formally as:

$$MongoRaftReconfig \Rightarrow \square StateMachineSafety \tag{4}$$

Note that the *MongoLoglessDynamicRaft* protocol also operates as a Raft based state machine, and it is responsible for safely managing the configuration state. So, an auxiliary correctness goal which we also verify is that *MongoLoglessDynamicRaft* satisfies the *StateMachineSafety* property, which we examine in more detail below, in Section 5.2.3.

5.2 Model Checking

We undertook an automated approach to verifying safety using TLC [46], an explicit state model checker for TLA+ specifications. We verified finite instances of the protocol to provide a sound guarantee of protocol correctness up to a certain size. It has been observed elsewhere [24] that relatively small, finite instances of distributed protocols are often sufficient to exhibit behaviors that are generalizable to larger (potentially infinite) instances, which provides confidence in our approach.

We automatically verified the *StateMachineSafety* invariant by model checking a finite instance of the *MongoRaftReconfig* specification. Verifying this specification, however, encountered scalability issues even for very small models. To alleviate this, we additionally verified the *MongoLoglessDynamicRaft* protocol in isolation, which allowed us to check finite models with significantly larger parameters. In Section 5.3 we show, by a refinement based argument, that any properties of *MongoLoglessDynamicRaft* hold in *MongoRaftReconfig*. This allows us to assume our verification efforts for *MongoLoglessDynamicRaft* hold in *MongoRaftReconfig*, providing stronger confidence in the correctness of the overall protocol.

5.2.1 The TLC Model Checker. TLC is an explicit state model checker that can check temporal properties of a given TLA+ specification. It is provided as a Java program that takes as input a TLA+ module file, a model checker configuration file, and a set of command line parameters. For checking safety properties, TLC assumes a TLA+ specification of the form $Init \land \square[Next]_{vars}$. The configuration file tells TLC the name of the specification to check and of the properties to be checked. In addition, the configuration file defines a *model* of the specification, which is an assignment of values to any constant parameters of the specification. It is also possible to provide a *state constraint*, which is a state predicate that can be used to constrain the set of reachable states. If TLC discovers a reachable state that violates the state constraint predicate, it will not add the state to its current graph of reachable states. TLC also allows definition of a *symmetry set*, which causes the model checker to consider states that have the same constant value under some permutation as equivalent, which can significantly reduce the set of reachable states for certain models [11]. A more complete and in-depth explanation of TLC behavior and parameters can be found in [18]. For all model checking runs discussed below we used TLC version 2.15 (adc67eb) running with a single worker thread on CentOS Linux 7, with a 2.30GHz Intel Xeon Gold 5118 CPU.

5.2.2 Model Checking of MongoRaftReconfig. For checking safety of *MongoRaftReconfig* we used a model we refer to as *MCMongoRaftReconfig*, which imposes finite bounds on the *MongoRaftReconfig* TLA+ specification. The complete, runnable TLC configuration for this model can be found in [37]. The model sets $Server = \{n1, n2, n3, n4\}$, and imposes

MCMongoRaftReconfig	
Server	$\{n1, n2, n3, n4\}$
MaxLogLen	2
MaxTerm	2
MaxConfigVersion	3
Constraint	*StateConstraint*
Symmetry	*Permutation(Server)*
Invariant	*StateMachineSafety*
States	18,955,578
Depth	29
Duration	6h 51min

(a)

MCMongoLoglessDynamicRaftAux	
Server	$\{n1, n2, n3, n4\}$
MaxTerm	4
MaxConfigVersion	4
Constraint	*StateConstraint*
Symmetry	*Permutation(Server)*
Invariant	*StateMachineSafety*
States	124,438,466
Depth	30
Duration	11h 35min

(b)

Fig. 3. Summary of TLC Model Checking Results. *States* is the number of reachable, distinct states discovered by TLC. *Depth* is the length of the longest behavior.

the following state constraint:

$$\begin{aligned} StateConstraint \triangleq \forall s \in Server : \\ &\land currentTerm[s] \leq MaxTerm \\ &\land Len(log[s]) \leq MaxLogLen \\ &\land configVersion[s] \leq MaxConfigVersion \end{aligned}$$

This constraint, along with a finite *Server* set, is sufficient to make the reachable state space of this model finite, since it limits the size of the three unbounded variables of the specification: *terms*, *logs*, and configuration *versions*. It restricts logs to be of a maximum finite length, and imposes a finite upper bound on terms and configuration versions. Figure 3a shows the parameters and results for this model. *Permutation* is an operator in the *TLC.tla* standard module [2] defined as the set of all permutations of elements in a given set. Under our symmetry declaration, any two states that are equal up to permutation of server identifiers are treated as equivalent by the model checker.

5.2.3 Model Checking of MongoLoglessDynamicRaft. Even with finite constraints and the use of symmetry optimizations, the complexity of the complete *MongoRaftReconfig* protocol limited the scalability of our verification efforts. As seen in Figure 3a, a model with 4 servers and $MaxLogLen = 2$, $MaxTerm = 2$, $MaxConfigVersion = 3$ produces over 18 million states and takes over 6 hours. To address this, we additionally verified the correctness of the *MongoLoglessDynamicRaft* subprotocol independently. *MongoLoglessDynamicRaft* operates its own replicated state machine and so must uphold the necessary safety properties in order for *MongoRaftReconfig* to operate safely. The compositional structure of *MongoRaftReconfig* makes it possible to verify *MongoLoglessDynamicRaft* in isolation and assume that its safety properties hold in *MongoRaftReconfig*. This technique allowed us to verify *MongoLoglessDynamicRaft* on models with significantly larger finite parameters and ensure that our results hold for *MongoRaftReconfig*. It provides stronger confidence that this subprotocol, which handles the main behaviors related to dynamic reconfiguration, is correct.

For model checking, we used an augmented version of the *MongoLoglessDynamicRaft* specification that we refer to as *MongoLoglessDynamicRaftAux*, whose complete definition is given in [38]. *MongoLoglessDynamicRaftAux* is a simple extension of *MongoLoglessDynamicRaft* that adds a *committed* variable, which is a *history variable* [21] that records the set of committed configurations as (v, t) pairs. Since the core *MongoLoglessDynamicRaft* protocol doesn't explicitly record committed configurations, this variable is necessary in order to state the *StateMachineSafety* property. Note that a history variable is a "passive" state variable that does not change the semantics of a specification i.e. it does not change a specification's set of behaviors. The full model checking results for our model *MCMongoLoglessDynamicRaftAux*, whose definition is provided in [36], are presented in Figure 3b. In the following section we show formally that it is sound to assume these results hold for *MongoRaftReconfig*, by showing that $MongoRaftReconfig \Rightarrow MongoLoglessDynamicRaft$.

5.3 Subprotocol Refinement

In order to ensure that the safety properties of *MongoLoglessDynamicRaft* hold for *MongoRaftReconfig*, we must demonstrate that the behaviors of *MongoLoglessDynamicRaft* are not augmented when operating as a subprotocol of *MongoRaftReconfig*. Formally, we want to show that $MongoRaftReconfig \Rightarrow MongoLoglessDynamicRaft$. This requires showing that, for any behavior σ of *MongoRaftReconfig*, the initial state of σ is a valid initial state of *MongoLoglessDynamicRaft* and every transition in σ is a valid transition of *MongoLoglessDynamicRaft.* For sake of brevity below, we use *MRR* and *MLDR*, respectively, as abbreviations for *MongoRaftReconfig* and *MongoLoglessDynamicRaft.* Formally, we must show

$$MRR!Init \Rightarrow MLDR!Init \tag{5}$$

$$[MRR!Next]_{vars_{MRR}} \Rightarrow [MLDR!Next]_{vars_{MLDR}} \tag{6}$$

where $vars_{MRR}$ and $vars_{MLDR}$ are, respectively, the variables of *MongoRaftReconfig* and *MongoLoglessDynamicRaft*, as summarized in Figure 2. For a specification S, the expressions $S!Init$ and $S!Next$ refer, respectively, to the initial state predicate and next state relation of S. Recall that $[N]_{vars} = N \lor \text{UNCHANGED } vars$ i.e. it is an action that allows for stuttering steps. We define $vars_{MSR} = \langle log, committed \rangle$ as the set of variables that are *private* to *MongoStaticRaft*, meaning they are not contained in $vars_{MLDR}$.

In order to prove Formula 5, it is sufficient to show that all initial states allowed by $MRR!Init$ satisfy $MLDR!Init$. As discussed in Section 4.3.1, the valid initial states of both *MRR* and *MLDR* are the same i.e. $currentTerm[s] = 0$, $state[s] = Secondary$, $configVersion[s] = 1$, $configTerm[s] = 0$, and $config[s] = m$ for all servers s and some member set m. In order to prove Formula 6, we must show that each transition allowed by $MRR!Next$ is a valid $MLDR!Next$ transition. For a TLA+ expression $N = A_1 \lor ... \lor A_n$, for actions A_i, we call each A_i a *subaction* of N. Note that for any subaction A_i of N, it holds that $A_i \Rightarrow N$. So, it is sufficient to show that, for each subaction A of $MRR!Next$, the following holds

$$A \Rightarrow MLDR!Next \lor \text{UNCHANGED } vars_{MLDR} \tag{7}$$

We first consider actions which modify only private variables $vars_{MSR}$. Each such action, A, is shown below as A^{vars}, where $vars$ is the list of variables that are modified by that action:

- $ClientRequest(s)^{\langle log \rangle}$
- $GetEntries(s, t)^{\langle log \rangle}$
- $RollbackEntries(s, t)^{\langle log \rangle}$
- $CommitEntry(s, Q)^{\langle committed \rangle}$

The actions above modify only the *log* or *committed* variable, so they trivially satisfy UNCHANGED $vars_{MLDR}$ i.e. they are stuttering steps of *MLDR*, which is sufficient to satisfy the implication of Formula 7. Next we examine the remaining actions:

- $SendConfig(s, t)$
- $OplogCommitment(s) \land Reconfig(s)$
- $BecomeLeader(s, Q) \land BecomeLeaderDynamic(s, Q)$
- $UpdateTerms(s, t) \land UpdateTermsDynamic(s, t)$

Each of these actions is of the form $P \land A_{MLDR}$, where A_{MLDR} is a subaction of $MLDR!Next$. We know that for any such subaction, $A_{MLDR} \Rightarrow MLDR!Next$. Since $P \land A_{MLDR} \Rightarrow A_{MLDR}$, it holds that $P \land A_{MLDR} \Rightarrow MLDR!Next$ which implies that Formula 7 holds for each action. Thus, we have proved Formula 6, which was our goal.

Establishing that $MRR \Rightarrow MLDR$ allows us to soundly assume that the safety properties of *MongoLoglessDynamicRaft* hold when operating as a subprotocol of *MongoRaftReconfig*. That is, for any safety property P such that $MLDR \Rightarrow P$, it holds, by transitivity of implication, that $MRR \Rightarrow P$. This allows us to assume that the safety properties of *MongoLoglessDynamicRaft* verified in Section 5.2.3 hold in *MongoRaftReconfig*. We do not formalize the interface between *MongoRaftReconfig* and *MongoLoglessDynamicRaft* here, but our verification efforts provide confidence in the safety of *MongoLoglessDynamicRaft*, the most important and novel subcomponent of *MongoRaftReconfig*.

6 COMPARISON TO STANDARD RAFT

6.1 Behavioral Mapping

MongoRaftReconfig can be viewed as a generalization and optimization of the standard Raft dynamic reconfiguration protocol. To show how our protocol relates to and extends standard Raft, we discuss the two primary aspects of the protocol which set it apart from Raft: (1) decoupling of the oplog and config log and (2) logless optimization of the config log.

6.1.1 Decoupling the Configuration Log. 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 become committed i.e. it has propagated to a quorum of servers. 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 two logs and their commitment rules, allowing reconfigurations to bypass the oplog if necessary. We examine this benefit experimentally in Section 6.2.

6.1.2 Logless Optimization. Decoupling the config log 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, which is implemented in *MongoLoglessDynamicRaft*, 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, which we refer to as *MergeEntries*. This action conceptually subsumes the *GetEntries* and *RollbackEntries* actions of the *MongoStaticRaft* specification described in Section 4.2, which is a log-based protocol. We do not formally define *MergeEntries* here, but it can be viewed as an action where one server s atomically transfers its entire log to another server t, if the log of s is newer, based on the index and term of its last entry. In *MongoLoglessDynamicRaft*, the *SendConfig* action implements this behavior to transfer configuration state between servers.

6.2 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*, however, reconfigurations bypass the oplog replication channel, committing without the need to commit writes in the oplog. This allows the protocol 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.

6.2.1 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. Our experiment [2] 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 [1] 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

[2] The source code used for our experiments is available upon request.

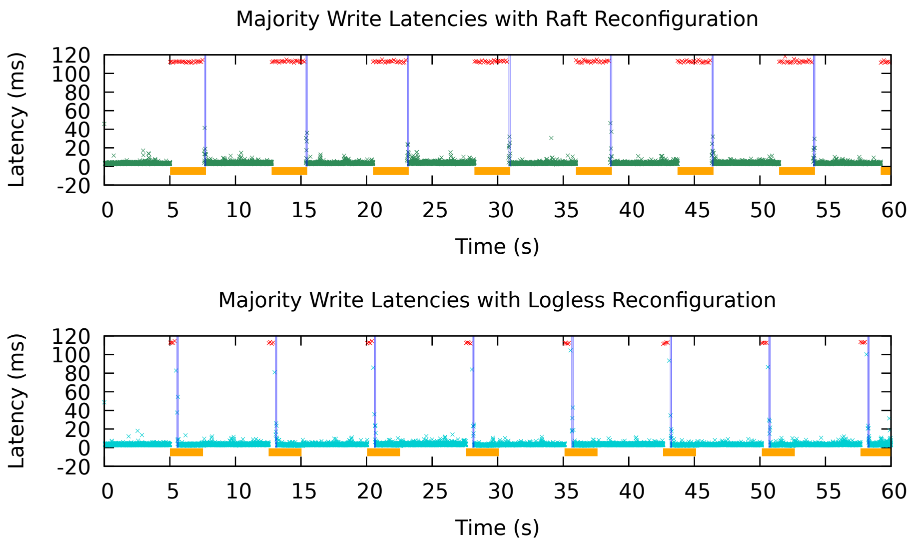

Fig. 4. Latency of majority writes in the face of node degradation and reconfiguration to recover. Red points indicate writes that timed out i.e. failed to commit. Orange horizontal bars indicate intervals of time where system entered a *degraded* mode. Thin, vertical blue bars indicate successful completion of reconfiguration events.

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 *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 4. 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 [17], 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 [25] 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 [20]. In [10], 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 [31], 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 [28]. Formal verification of Raft’s linearizability guarantees was later completed in Verdi [45], a framework for verifying distributed systems in the Coq proof assistant [7], 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 [6], 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 [29], but the project was never extended to include the fixed version of the protocol. The Zab replication protocol, implemented in Apache Zookeeper [42], 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 not based on Raft. RAMBO [13], 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 [44] 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 [15], and approaches to logless implementations of Paxos based replicated state machine protocols [34], which bear conceptual similarities to our logless protocol for managing configuration state.

Our formal specification and verification efforts follow prior lines of work on formally verifying distributed protocols e.g. Paxos and its variants [9, 19], the Chord [23] protocol, the Pastry distributed hash table [22], and others [8, 27]. Distributed protocols are subtle and challenging to design correctly, so they benefit greatly from precise, machine checkable descriptions. More recent progress has also been made on tools to help automate the verification and proof of protocols like these even further e.g. Ivy [32] and I4 [24].

REFERENCES

[1] 2020. MongoDB JIRA SERVER-46907. https://jira.mongodb.org/browse/SERVER-46907

[2] 2020. *TLC.tla Module.* https://github.com/tlaplus/tlaplus/blob/master/tlatools/org.lamport.tlatools/src/tla2sany/StandardModules/TLC.tla

[3] 2021. MongoDB Github Project. https://github.com/mongodb/mongo

[4] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* (1991). https://doi.org/10.1016/0304-3975(91)90224-P

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

[6] Brandon Amos and Huanchen Zhang. 2015. *Specifying and proving cluster membership for the Raft distributed consensus algorithm.* Technical Report. https://www.cs.cmu.edu/~aplatzer/course/pls15/projects/bamos.pdf

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

[8] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. 2020. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[9] Saksham Chand, Yanhong A Liu, and Scott D Stoller. 2016. Formal verification of multi-Paxos for distributed consensus. In *International Symposium on Formal Methods*. Springer, 119–136.

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

[11] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. 1998. Symmetry reductions in model checking. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/bfb0028741

[12] 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. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*. https://doi.org/10.1145/2518037.2491245

[13] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. 2010. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing* (2010). https://doi.org/10.1007/s00446-010-0117-1

[14] 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. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* (2020). https://doi.org/10.14778/3415478.3415535

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

[16] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* (1998). https://doi.org/10.1145/279227.279229

[17] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* (2001). https://doi.org/10.1145/568425.568433

[18] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/

[19] Leslie Lamport. 2011. Byzantizing Paxos by refinement. In *International Symposium on Distributed Computing*. Springer, 211–224.

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

[21] Leslie Lamport and Stephan Merz. 2017. Auxiliary variables in TLA+. arXiv:1703.05121

[22] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. 2011. Towards Verification of the Pastry Protocol Using TLA + . In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–258.

[23] Jørgen Aarmo Lund. 2019. *Verification of the Chord protocol in TLA+*. Master's thesis. UiT Norges arktiske universitet.

[24] Haojun Ma, Aman Goel, Jean Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles*. https://doi.org/10.1145/3341301.3359651

[25] Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report MSR-TR-2008-192. https://www.microsoft.com/en-us/research/publication/stoppable-paxos/

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

[27] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2014. Use of formal methods at Amazon Web Services. *See http://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf* (2014).

[28] Diego Ongaro. 2014. Consensus: Bridging Theory and Practice. *Doctoral thesis* (2014).

[29] Diego Ongaro. 2015. Bug in single-server membership changes. https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J.

[30] Diego Ongaro. 2021. The Raft Consensus Algorithm. https://raft.github.io/

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

[32] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[33] Amir Pnueli. 1977. The Temporal Logic of Programs. (1977).

[34] Denis Rystsov. 2018. CASPaxos: Replicated State Machines without logs. arXiv:1802.07000

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

[36] William Schultz. 2021. *MCMongoLoglessDynamicRaftAux TLC Model*. https://github.com/will62794/logless-reconfig/blob/63edb2573cb9fd38f681890283235aaf6cd320e7/specs/models/MCMongoLoglessDynamicRaftAux-4Servers-T4-CV4.cfg

[37] William Schultz. 2021. *MCMongoRaftReconfig TLC Model.* https://github.com/will62794/logless-reconfig/blob/63edb2573cb9fd38f681890283235aaf6cd320e7/specs/models/MCMongoRaftReconfig-4Servers-L2-T2-CV3.cfg

[38] William Schultz. 2021. *MongoLoglessDynamicRaftAux TLA+ Specification.* https://github.com/will62794/logless-reconfig/blob/63edb2573cb9fd38f681890283235aaf6cd320e7/specs/MongoLoglessDynamicRaftAux.tla

[39] William Schultz. 2021. *MongoRaftReconfig TLA+ Specification.* https://github.com/will62794/logless-reconfig/blob/63edb2573cb9fd38f681890283235aaf6cd320e7/specs/MongoRaftReconfig.tla

[40] William Schultz. 2021. *MongoStaticRaft TLA+ Specification.* https://github.com/will62794/logless-reconfig/blob/63edb2573cb9fd38f681890283235aaf6cd320e7/specs/MongoStaticRaft.tla

[41] William Schultz, Tess Avitabile, and Alyson Cabral. 2018. Tunable consistency in MongoDB. In *Proceedings of the VLDB Endowment.* https://doi.org/10.14778/3352063.3352125

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

[43] 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. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20).* Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

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

[45] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. 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.* https://doi.org/10.1145/2854065.2854081

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

[47] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: High-availability storage made practical in WeChat. In *Proceedings of the VLDB Endowment.* https://doi.org/10.14778/3137765.3137778

[48] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21).* {USENIX} Association. https://www.usenix.org/conference/nsdi21/presentation/zhou