Formal Verification of a Distributed Dynamic Reconfiguration Protocol*

William Schultz†
Northeastern University
Boston, MA, USA
schultz.w@northeastern.edu

Ian Dardik†
Northeastern University
Boston, MA, USA
dardik.i@northeastern.edu

Stavros Tripakis
Northeastern University
Boston, MA, USA
stavros@northeastern.edu

Abstract

We present a formal, machine checked TLA+ safety proof of *MongoRaftReconfig*, a distributed dynamic reconfiguration protocol. *MongoRaftReconfig* was designed for and implemented in MongoDB, a distributed database whose replication protocol is derived from the Raft consensus algorithm. We present an inductive invariant for *MongoRaftReconfig* that is formalized in TLA+ and formally proved using the TLA+ proof system (TLAPS). We also present a formal TLAPS proof of two key safety properties of *MongoRaftReconfig*, *LeaderCompleteness* and *StateMachineSafety*. To our knowledge, these are the first machine checked inductive invariant and safety proof of a dynamic reconfiguration protocol for a Raft based replication system.

CCS Concepts: • **Theory of computation → Automated reasoning**; • **Computing methodologies → Distributed algorithms**.

Keywords: Formal Verification, Theorem Proving, TLA+, Dynamic Reconfiguration, Distributed Systems, Raft

ACM Reference Format:
William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3497775.3503688

*This work has been partially supported by NSF award CNS-1801546.
†Both authors contributed equally to this research.

CPP '22, January 17–18, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9182-5/22/01…$15.00
https://doi.org/10.1145/3497775.3503688

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 [7, 8, 16, 41]. Paxos [19] and, more recently, Raft [29], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [1] for the protocols underlying these systems.

Furthermore, few of these reconfiguration protocols have been formally verified [26, 34, 39]. 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 [28] 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. This also demonstrates that formal verification is valuable for ensuring correctness of these protocols.

MongoDB [25] is a general purpose, document oriented database which implements a distributed replication system [36] for providing high availability and fault tolerance. MongoDB's replication system uses a log-based consensus protocol that derives from Raft [47]. MongoDB recently introduced a novel dynamic reconfiguration protocol, *MongoRaftReconfig*, for its replication system. The *MongoRaftReconfig* protocol is described in detail in [38], which includes a TLA+ formal specification of the protocol and a manual safety proof.

In this paper, we present the first formal verification of the safety properties of *MongoRaftReconfig*. We present a formally stated inductive invariant for the protocol, which we prove and then utilize to establish two high level safety properties of the protocol. In particular, we prove (1) *LeaderCompleteness*, which, intuitively, states that if a log entry is committed it is durable, and (2) *StateMachineSafety*, which says that log entries committed at a particular index must be

consistent across all nodes in the system. We carry out our verification efforts using TLAPS, the TLA+ proof system [9]. To our knowledge, this is the first machine checked inductive invariant and safety proof of a reconfiguration protocol for a Raft based replication system.

To summarize, we make the following contributions:

- A formally stated TLA+ inductive invariant for the *MongoRaftReconfig* protocol. To our knowledge, this is both the first inductive invariant for a Raft-based reconfiguration protocol and the first that has been formalized.
- A formally verified TLAPS proof of our inductive invariant.
- A formally verified TLAPS proof that *MongoRaftReconfig* satisfies the two above safety properties, *LeaderCompleteness* and *StateMachineSafety*. To our knowledge, this is the first machine checked safety proof of a Raft-based reconfiguration protocol.

All of our TLA+ specifications, TLAPS proof code, and instructions for checking our proofs are included in the supplementary material [37] for this paper. Where appropriate throughout this paper, we cite the relevant files located in this material.

The rest of this paper is organized as follows. In Section 2 we provide some general background about MongoDB replication and TLA+. Section 3 provides a formal statement of the verification results that we establish in this paper. Section 4 presents our inductive invariant for *MongoRaftReconfig*, and Section 5 presents our formal safety proof of *MongoRaftReconfig* in TLAPS, which makes use of our inductive invariant. Section 6 discusses related work, and Section 7 discusses conclusions and future work.

2 Background

2.1 The MongoDB Static Replication Protocol

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 [27], with some extensions. We refer to MongoDB's abstract replication protocol, without reconfiguration, as *MongoStaticRaft*, to distinguish it from the *MongoRaftReconfig* protocol verified in this paper. *MongoStaticRaft* can be viewed as a modified version of standard Raft that satisfies the same underlying correctness properties, and it is described in more detail in [36, 47]. We provide a high level overview here, since *MongoRaftReconfig* is built on top of *MongoStaticRaft*.

A MongoDB replica set running *MongoStaticRaft* consists of a set of server processes, $Server = \{s_1, s_2, \ldots, s_n\}$. 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.

2.2 The MongoDB Dynamic Reconfiguration Protocol: MongoRaftReconfig

MongoRaftReconfig, the protocol verified in this paper, is an extension of *MongoStaticRaft* that allows for dynamic reconfiguration. *MongoRaftReconfig* utilizes a logless approach to managing configuration state and decouples the processing of configuration changes from the main database operation log. The full details of *MongoRaftReconfig* are presented in [38], but we provide a high level overview here. In *MongoRaftReconfig*, each server of a replica set maintains a single, durable *configuration*, where a configuration is formally defined as a tuple (m, v, t), where $m \in 2^{Server}$ is a subset of all servers, $v \in \mathbb{N}$ is a numeric configuration *version*, and $t \in \mathbb{N}$ is the numeric *term* of the configuration. Configurations are totally ordered by their $(version, term)$ pair, where $term$ is compared first, followed by version. Servers can install any configuration newer than their own. Reconfiguration operations, which can only be processed by primary servers, update a server's local configuration to a new configuration specified by the client.

Prior to the work presented in this paper, [38] presented a formal TLA+ specification of *MongoRaftReconfig*, results from model checking its safety on finite protocol instances, and a manual, prose safety proof. There existed, however, no formal inductive invariant or machine checked proof for the safety properties of *MongoRaftReconfig*. The formal inductive invariant we present in this paper bears some structural similarity to the manual proof given in [38], but the TLAPS proofs presented in this paper were developed independently, and are not based on the prior, manual proof.

2.3 TLA+ and TLAPS

TLA+ [20] is a formal specification language for describing distributed and concurrent systems that is based on first order and temporal logic [33]. Since *MongoRaftReconfig* is formally specified using the TLA+ language and it is the

language used for our proofs, we provide a brief overview of TLA+ and its associated proof system, TLAPS [9].

2.3.1 Specifications in TLA+. 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, and $vars$ denotes a sequence of all state variables of a specification. $[Next]_{vars}$ 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 TLA+ expression containing state variables, expressed by attaching a $'$ symbol, denotes the value of that expression 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. Correctness properties and system specifications in TLA+ are both written as temporal logic formulas. This allows one to express notions of property satisfaction in a concise manner. We say that a specification S *satisfies* a property P iff the formula $S \Rightarrow P$ is valid (i.e. true under all assignments).

2.3.2 The TLA+ Proof System. The TLA+ proof system [9], abbreviated as TLAPS, is an accompanying tool for the TLA+ language that allows one to write and mechanically check hierarchically structured proofs [18] in TLA+. Proofs consist of a series of statements that support the proof goal, which is the top level statement that must be proved. Each statement, in turn, must be proved either compositely using a nested structural proof, or as a leaf proof via a backend solver. TLAPS is independent of any particular SMT solver or theorem prover, and includes support for various backends e.g. Z3 [11], Isabelle [43], and Zenon [4].

Figure 1 shows an example of a lemma and its proof in TLAPS. The ASSUME-PROVE idiom treats the lemma as an implication. That is, if *Conditions* hold, then *Implication* must follow. Leaf statements are proved using the BY statement, and can reference theorems and lemmas by name, operator definitions, and previous statements by label. Each structural proof must end with a QED statement, closing the goal of either a nested or overall proof.

2.4 The MongoRaftReconfig TLA+ Specification

A formal TLA+ specification of *MongoRaftReconfig* was originally included in [38], but was not discussed in detail. This same specification serves as the basis for the TLAPS proofs

```
LEMMA NameOfLemma ≜
ASSUME Conditions
PROVE Implication
PROOF
  ⟨1⟩1. Statement1.1 BY DEF Conditions
  ⟨1⟩2. Statement1.2
    ⟨2⟩1. Statement2.1 BY UsefulTheorem
    ⟨2⟩2. Statement2.2 BY ⟨1⟩1, ⟨2⟩1
    ⟨2⟩. QED BY ⟨2⟩2
  ⟨1⟩. QED BY ⟨1⟩1, ⟨1⟩2
```

Figure 1. Example of a hierarchically structured TLAPS proof.

$$
\begin{array}{ll}
TypeOK \triangleq & \\
\quad log & \in [Server \to Seq(\mathbb{N})] \\
\quad committed & \in 2^{\mathbb{N}\times\mathbb{N}} \\
\quad term & \in [Server \to \mathbb{N}] \\
\quad state & \in [Server \to \{Primary, Secondary\}] \\
\quad config & \in [Server \to 2^{Server}] \\
\quad configVersion & \in [Server \to \mathbb{N}] \\
\quad configTerm & \in [Server \to \mathbb{N}]
\end{array}
$$

Figure 2. The state variables of the *MongoRaftReconfig* protocol and their corresponding types stated as a type correctness predicate in TLA+. The notation $[A \to B]$ represents the set of all functions from set A to set B and $Seq(S)$ represents the set of all sequences containing elements from the set S.

presented in this paper, so we give a brief overview of the specification here. The complete specification can be found in the *MongoRaftReconfig.tla* file of the supplementary material provided with this paper [37].

The state variables of the specification and their types are shown in Figure 2. The initial states, next state relation, and specification definition of *MongoRaftReconfig* are summarized in Figure 3. The operator $Quorums(m)$ is defined as the set of all majority quorums [42] for a given set of servers m. Reconfigurations are modeled by the $Reconfig(s, m)$ action, which represents a reconfiguration that occurs on primary server s to a new configuration with member set $m \in 2^{Server}$. Configuration propagation is modeled by the $SendConfig(s, t)$ action, which represents the propagation of a configuration from server s to server t. Elections are modeled by the action $BecomeLeader(s, Q)$, which represents the election of server s by a set of voters Q. The action $UpdateTerms(s, t)$ propagates the term of a server s to server t, if the term of s is newer than t. The actions $ClientRequest(s)$, $GetEntries(s, t)$, $RollbackEntries(s, t)$, and $CommitEntry(s, Q)$ are responsible for log related actions that are conceptually unrelated to reconfiguration, so we do not discuss their details here. Their full definitions can

MODULE *MongoRaftReconfig*

$$
\begin{aligned}
&MRRInit \triangleq \\
&\quad \land\ log = [i \in Server \mapsto \langle\rangle] \\
&\quad \land\ committed = \{\} \\
&\quad \land\ currentTerm = [i \in Server \mapsto 0] \\
&\quad \land\ state = [i \in Server \mapsto Secondary] \\
&\quad \land\ configVersion = [i \in Server \mapsto 1] \\
&\quad \land\ configTerm = [i \in Server \mapsto 0] \\
&\quad \land\ \exists\, initConfig \in 2^{Server} : \\
&\qquad \land\ initConfig \neq \emptyset \\
&\qquad \land\ config = [i \in Server \mapsto initConfig] \\
&\\
&MRRNext \triangleq \\
&\quad \exists\, s,\ t \in Server : \\
&\quad \exists\, Q \in Quorums(config[s]) : \\
&\qquad \lor\ ClientRequest(s) \\
&\qquad \lor\ GetEntries(s,\ t) \\
&\qquad \lor\ RollbackEntries(s,\ t) \\
&\qquad \lor\ CommitEntry(s,\ Q) \\
&\qquad \lor\ SendConfig(s,\ t) \\
&\qquad \lor\ Reconfig(s, m) \\
&\qquad \lor\ BecomeLeader(s,\ Q) \\
&\qquad \lor\ UpdateTerms(s,\ t) \\
&\\
&MRRSpec \triangleq MRRInit \land \Box[MRRNext]_{vars}
\end{aligned}
$$

Figure 3. Summary of the *MongoRaftReconfig* TLA+ specification. The full specification consists of 359 lines of TLA+ code, excluding comments, and can be found in the *MongoRaftReconfig.tla* file of the supplementary material.

be found in the specifications provided in the supplementary material [37].

Note that our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol. 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 our automated verification and proof efforts more feasible. In the future, however, we believe it would be valuable to explore techniques for formally relating our abstract specifications to real world protocol implementations, with an aim of verifying whether a system implementation faithfully reflects our high level specifications [5, 10, 15].

3 Verification Problem Statement

In this paper we establish that the *MongoRaftReconfig* protocol satisfies *LeaderCompleteness* and *StateMachineSafety*, which are two key, high level safety properties of both the MongoDB replication system and standard Raft. Informally, the *LeaderCompleteness* property states that if a log entry is committed in term T, then it is present in the log of any leader in term $T' > T$. It is stated more precisely in Definition 3.1, 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. *StateMachineSafety* states that if two log entries are committed at the same log index, these entries must be the same, and is stated formally as Definition 3.2.

Definition 3.1 (Leader Completeness)**.**

$$
\begin{aligned}
&\forall s \in Server : \forall (cindex, cterm) \in committed : \\
&\quad (state[s] = Primary \land cterm < term[s]) \Rightarrow \\
&\quad InLog(cindex, cterm, s)
\end{aligned}
$$

Definition 3.2 (State Machine Safety)**.**

$$
\begin{aligned}
&\forall (ind_i, t_i), (ind_j, t_j) \in committed : \\
&\qquad (ind_i = ind_j) \Rightarrow (t_i = t_j)
\end{aligned}
$$

Both *LeaderCompleteness* and *StateMachineSafety* are safety properties. More specifically, they are both invariants, meaning that they must hold in all reachable states of *MongoRaftReconfig*. Thus, our verification goals can be stated formally as Theorems 3.3 and 3.4, where $MRRSpec$ refers to the specification of *MongoRaftReconfig* as given in Figure 3.

Theorem 3.3 (MRRImpliesLeaderCompleteness)**.**

$$MRRSpec \Rightarrow \Box LeaderCompleteness$$

Theorem 3.4 (MRRImpliesStateMachineSafety)**.**

$$MRRSpec \Rightarrow \Box StateMachineSafety$$

Theorems 3.3 and 3.4 are the safety results established and formally verified in this paper, and they can be found in the *MongoRaftReconfigProofs.tla* file of our supplementary material [37]. The proofs of Theorems 3.3 and 3.4 are discussed in Section 5. Both of these theorems are proved using the help of an inductive invariant, which we discuss next, in Section 4.

4 The Inductive Invariant

4.1 Background

A standard method to establish an invariant Inv is to find an *inductive invariant* that implies Inv [24]. Formally, a state predicate Inv is an invariant of a system $Spec$ if the following holds:

$$Spec \Rightarrow \Box Inv \tag{1}$$

Suppose that $Spec$ is of the form $Spec = Init \land \Box[Next]_{vars}$, as in the case of *MongoRaftReconfig*. Then, in order to establish Formula 1, it is sufficient to find a state predicate Ind

$$
\begin{array}{ll}
MRRInd \triangleq & \\
T & \wedge\ TypeOK \\
E_1 & \wedge\ ElectionSafety \\
 & \wedge\ PrimaryConfigTermEqualToCurrentTerm \\
 & \wedge\ ConfigVersionAndTermUnique \\
 & \wedge\ PrimaryInTermContainsNewestConfigOfTerm \\
 & \wedge\ ActiveConfigsOverlap \\
 & \wedge\ ActiveConfigsSafeAtTerms \\
L_1 & \wedge\ LogEntryInTermImpliesConfigInTerm \\
 & \wedge\ PrimaryHasEntriesItCreated \\
 & \wedge\ LogMatching \\
L_2 & \wedge\ PrimaryTermAtLeastAsLargeAsLogTerms \\
 & \wedge\ TermsOfEntriesGrowMonotonically \\
 & \wedge\ UniformLogEntriesInTerm \\
C_1 & \wedge\ CommittedEntryIndexesAreNonZero \\
 & \wedge\ CommittedTermMatchesEntry \\
C_2 & \wedge\ LeaderCompleteness \\
 & \wedge\ LogsLaterThanCommittedMustHaveCommitted \\
 & \wedge\ ActiveConfigsOverlapWithCommittedEntry \\
 & \wedge\ NewerConfigsDisableCommitsInOlderTerm \\
N & \wedge\ ConfigsNonEmpty
\end{array}
$$

Figure 4. Our inductive invariant for *MongoRaftReconfig*.

such that the following conditions hold:

$$Init \Rightarrow Ind \tag{2}$$

$$Ind \wedge Next \Rightarrow Ind' \tag{3}$$

$$Ind \Rightarrow Inv \tag{4}$$

Conditions 2 and 3 are referred to as *initiation* and *consecution*, respectively, and they are sufficient to show that Ind is an inductive invariant. Conditions 2, 3, and 4 are together sufficient to establish Formula 1.

In our case, $Spec$ instantiates to $MRRSpec$ and we have two instances of Inv, namely, *LeaderCompleteness* and *StateMachineSafety*. In principle, we need to discover two distinct inductive invariants, one for *LeaderCompleteness* and another one for *StateMachineSafety*. In our case, the same inductive invariant turns out to be sufficient for both properties.

4.2 Invariant Overview

The inductive invariant that we developed for *MongoRaftReconfig* is referred to as $MRRInd$ and consists of 20 high level conjuncts, shown in Figure 4. Its full definition is given in 140 lines of TLA+ code, and is provided in the *MongoRaftReconfigIndInv.tla* file of our supplementary material [37].

The inductive invariant, shown in Figure 4, is composed of several conceptually distinct subcomponents. The first conjunct, $TypeOK$, establishes basic type-correctness constraints on the state variables of *MongoRaftReconfig*. This is necessary in most cases when stating inductive invariants in TLA+, since it is an untyped formalism [23]. The full definition of $TypeOK$ is shown in Figure 2. The initial set of 6 conjuncts, labeled as E_1 in Figure 4, along with $TypeOK$, is itself an inductive invariant, and it establishes the $ElectionSafety$ property, a key auxiliary invariant of the protocol that is needed to establish $LeaderCompleteness$. The conjuncts in group L_1 are a set of invariants related to logs of servers in the system, and they collectively establish $LogMatching$, another important auxiliary invariant. The L_2 group establishes a few additional log related invariants, which rely on previous conjuncts. In general, these log related conjuncts are not fundamentally related to dynamic reconfiguration, but are necessary to state precisely for a protocol that manages logs in a Raft like fashion. Group C_1 establishes some required, trivial aspects of the set of committed log entries. The conjunct group C_2 establishes the high level $LeaderCompleteness$ property, by relating how configurations interact with the set of committed log entries present in the system. Finally, the last conjunct, labeled as N, asserts that every configuration is non empty i.e. it contains some servers. This is an auxiliary invariant that is helpful for proving other facts.

4.3 Discovering an Inductive Invariant

Discovering such an inductive invariant for a protocol of this complexity is non-trivial. To our knowledge, this is the first inductive invariant proposed for a dynamic reconfiguration protocol that is built on a Raft based replication system. The discovery of $MRRInd$ took approximately 1-2 human months of work and it involved repeated efforts of iteration and refinement. To aid in this discovery process, we leveraged a technique proposed in [22] that utilizes the TLC explicit state model checker [46] to probabilistically verify candidate inductive invariants. If a candidate Inv is not inductive, the TLC model checker can, with some probability, report a *counterexample to induction*. A *counterexample to induction* is a state transition $s \rightarrow t$ satisfying $MRRNext$, where s satisfies Inv and t violates Inv. These counterexamples are helpful to understand why a candidate invariant fails to be inductive, and how it may need to be modified or strengthened further. This probabilistic method can only be used on finite protocol instances, and it does not provide a proof that an invariant is inductive. Nevertheless, the technique proved to be highly effective, as it helped us to discover an inductive invariant that we eventually proved formally correct using TLAPS, as discussed more in Section 5. Furthermore, we did not discover any errors in our inductive invariant during the TLAPS proof process.

Note that, although having a tool for finding counterexamples to induction is helpful for finding errors in candidate inductive invariants, it still does not provide much guidance in developing an inductive invariant from scratch. That is, it does not necessarily provide a systematic methodology for converging to a correct inductive invariant. Rather, development of our inductive invariant still required a large amount of creativity and human reasoning, largely driven by strong prior intuitions about the correctness of the protocol. For example, rather than aiming to develop the entire inductive invariant at once, we were able to develop it in components, based partially on human intuition about certain auxiliary lemmas that we knew must hold true of the overall protocol. For example, the *ElectionSafety* and the *LogMatching* invariants (shown in Figure 4) are two such lemmas that we worked on establishing first, before moving on to discover the additional conjuncts needed to establish the *LeaderCompleteness* property.

5 TLAPS Proofs

In this section we present an overview of our formally verified safety proof of *MongoRaftReconfig*, which was completed using TLAPS, the TLA+ proof system [9]. Section 5.1 gives an overview of the proof that $MRRInd$ is an inductive invariant of *MongoRaftReconfig*, and Section 5.2 describes how this fact is used to prove *LeaderCompleteness* and *StateMachineSafety*, which are the key, high level safety properties that were defined in Section 3.

5.1 TLAPS Proof of Inductive Invariant

To establish that $MRRInd$ is an inductive invariant, we must prove that $MRRInd$ satisfies both the initiation (2) and consecution (3) conditions for *MongoRaftReconfig*, as described in Section 4. This is captured in Lemma 5.1.

Lemma 5.1 ($MRRInd$ is an inductive invariant)**.**

$$MRRInit \Rightarrow MRRInd \tag{a}$$
$$MRRInd \wedge MRRNext \Rightarrow MRRInd' \tag{b}$$

Cases (a) and (b) of Lemma 5.1 represent, respectively, initiation and consecution. The initiation case of Lemma 5.1 follows in a straightforward manner from the definitions of $MRRInit$ and $MRRInd$. Proving the consecution case of Lemma 5.1, however, is the most difficult and time consuming aspect of the verification efforts presented in this paper. At a high level, this proof consists of showing that, assuming $MRRInd$ holds in a current state, every transition of the protocol upholds $MRRInd$ in the next state. To break this verification problem into smaller steps, we decompose the proof first by each conjunct of $MRRInd$, and then we decompose by each protocol transition.

Specifically, consider the definition of $MRRInd$, which is composed of 20 conjuncts (as shown in Figure 4):

$$MRRInd \triangleq I_1 \wedge I_2 \wedge \ldots \wedge I_{20}$$

Our first decomposition step breaks down case (b) of Lemma 5.1 into the following, independent proof goals, one for each conjunct of $MRRInd$:

$$\begin{aligned} MRRInd \wedge MRRNext &\Rightarrow I_1' \\ MRRInd \wedge MRRNext &\Rightarrow I_2' \\ &\vdots \\ MRRInd \wedge MRRNext &\Rightarrow I_{20}' \end{aligned} \tag{5}$$

Furthermore, $MRRNext$ is the disjunction of eight protocol actions (as shown in Figure 3):

$$MRRNext \triangleq A_1 \vee A_2 \vee \cdots \vee A_8 \tag{6}$$

So, we further decompose each goal of Statement 5 into one case for each protocol action. That is, we decompose each goal $MRRInd \wedge MRRNext \Rightarrow I_j'$ into the following proof goals:

$$\begin{aligned} MRRInd \wedge A_1 &\Rightarrow I_j' \\ MRRInd \wedge A_2 &\Rightarrow I_j' \\ &\vdots \\ MRRInd \wedge A_8 &\Rightarrow I_j' \end{aligned} \tag{7}$$

Our proof follows this methodology for every conjunct of $MRRInd$ and every action of $MRRNext$. This produces a set of proof goals whose size is the product of the number of protocol actions (8) and the number of invariant conjuncts (20), totaling $8 * 20 = 160$ proof goals. This decomposition allowed us to focus on proving one, small goal at a time, while incrementally building a library of reusable lemmas. The TLAPS proof of Lemma 5.1 can be found in the *MongoRaftReconfigProofs.tla* file of our supplementary material [37], while our library of lemmas can be found in the *Lib.tla*, *BasicQuorumsLib.tla*, and *LeaderCompletenessLib.tla* files.

5.2 TLAPS Proof of Safety

Lemma 5.1 establishes that *MRRInd* is an inductive invariant of *MongoRaftReconfig*. In this section we provide an overview of our proofs for establishing that *MongoRaftReconfig* satisfies *LeaderCompletness* and *StateMachineSafety* (Theorems 3.3 and 3.4), which utilize $MRRInd$. We do this by establishing lemmas 5.2 and 5.3, which, together with Lemma 5.1, are sufficient to establish Theorems 3.3 and 3.4.

Lemma 5.2.

$$MRRInd \Rightarrow LeaderCompleteness$$

Lemma 5.3 (IndImpliesStateMachineSafety)**.**

$$MRRInd \Rightarrow StateMachineSafety$$

LeaderCompleteness is a conjunct of $MRRInd$ so the implication of Lemma 5.2 follows trivially. The proof of Lemma 5.3, which can be found in the *StateMachineSafetyLemmas.tla* file of our supplementary material [37], is not trivial and we present it in the following section as a concrete example of a TLAPS proof.

5.3 Example of a TLAPS Proof

In this section we present the proof of Lemma 5.3 to serve as an example of TLAPS. The proof relies on one additional lemma, stated as Lemma 5.4. The proof of Lemma 5.4 is contained in the *StateMachineSafetyLemmas.tla* file of the supplementary material [37].

Lemma 5.4 (CommitsAreLogEntries)**.**

$$\begin{aligned} &MRRInd \Rightarrow \\ &\forall c \in committed : \exists s \in Server : \\ &\qquad InLog(c.entry, s) \end{aligned}$$

The TLAPS proof of Lemma 5.3 is shown in Figure 5. The proof uses the ASSUME-PROVE idiom to show that *MRRInd* implies *StateMachineSafety*. By the definition of the *StateMachineSafety* property, we can establish the proof goal given in $\langle 1 \rangle 1$. Steps $\langle 1 \rangle 2$ and $\langle 1 \rangle 3$ assume that $c1$ and $c2$ are arbitrary committed entries that share the same index but are not identical, and PROVE FALSE OBVIOUS establishes that these assumptions will lead to a contradiction. $\langle 1 \rangle 4$ is a composite proof that shows that $c1$ and $c2$ cannot share the same term. Steps $\langle 1 \rangle 5$ through $\langle 1 \rangle 8$ use Lemma 5.4 to show that there exist servers $s1$ and $s2$ that respectively contain the committed entries $c1$ and $c2$ in their logs. The two cases $\langle 1 \rangle 9$ and $\langle 1 \rangle 10$ show that if either $c1$ or $c2$ has a larger term than the other, then we derive a contradiction as expected. Finally, it suffices to only consider cases $\langle 1 \rangle 9$ and $\langle 1 \rangle 10$ because of step $\langle 1 \rangle 4$, and hence the proof is complete.

5.4 Proof Statistics

We now present some summary statistics about our TLAPS proof and its development to give a better sense of its scope, size, and difficulty. The entire TLAPS proof, including the statement of the inductive invariant and the protocol specification, consists of 3189 lines of TLA+ code, excluding comments. 140 of these lines are used for defining the inductive invariant and 359 of these lines are used for specifying the *MongoRaftReconfig* protocol. There are a total of 3 top level theorems and 78 formally stated lemmas. In terms of proof effort, we spent approximately 4 human-months on development of the TLAPS proof, which does not include the time to develop the inductive invariant described in Section 4. Development of the inductive invariant took approximately an additional 1-2 human-months of work. For the TLAPS proof system to check the correctness of the completed proof from scratch it takes approximately 38 minutes on a 2020 Macbook Air using 8 Apple M1 CPU Cores. This computation time consists mostly of queries to an underlying backend solver e.g. Isabelle or an SMT solver.

5.5 Experience with TLAPS

The hierarchical structure enforced by TLAPS led to well organized and generally readable proofs in our experience. Despite our overall positive experience, there two main shortcomings of TLAPS that we highlight below.

First, TLAPS does not offer much guidance when a backend solver fails on a leaf proof. In general, TLAPS does not distinguish between obligations that fail because they are false, versus obligations that are too difficult for the backend solvers. Second, we found that the TLAPS library did not always cater to our needs as conveniently as we hoped. For example, the *MongoRaftReconfig* specification includes state variables that are represented as TLA+ sequences, which are indexed using $\mathbb{N} \setminus \{0\}$. While the TLAPS standard library has theorems for induction on $\mathbb{N}$ (*NaturalsInduction.tla*), we were not able to find direct support for induction over the domain of sequences. Support for induction over the domain of sequences was not seamless, yet we were able to prove the desired theorem by tailoring parts of the library to our needs.

5.6 Discussion

Formally verifying safety properties for a large, real world distributed protocol is, in our experience, a very labor intensive task. Even if one has built up strong intuitions about correctness of a protocol, verification may take several months. Nevertheless, we believe that formal verification is of great value since, even for protocols that have been formally specified or model checked, design errors are still possible. For example, a safety bug in EPaxos [26], a well known variant of the original Paxos protocol, was discovered several years after its initial publication [40], even though EPaxos was accompanied by a TLA+ specification and manual safety proof in its original publication. Similarly, a bug in one of Raft's original reconfiguration protocols was also discovered after initial publication [28].

Furthermore, developing a formal inductive invariant and safety proof often provides deeper insights into why a protocol is correct, which fully automated techniques like model checking, on their own, are often unable to provide. Gaining deeper, formalized understanding of why a protocol is correct is valuable both from a theoretical perspective and also for system designers and engineers who may implement these protocols with extensions, modifications, or optimizations.

6 Related Work

Previously, there have been a variety of distributed protocols formalized using TLAPS, including Classic Paxos [6], Byzantine Paxos [21], and the Pastry distributed hash table protocol [2].

```
LEMMA IndImpliesStateMachineSafety ≜
ASSUME MRRInd
PROVE StateMachineSafety
  ⟨1⟩0. TypeOK BY DEF MRRInd
  ⟨1⟩1. SUFFICES ∀ c1, c2 ∈ committed :
        (c1.entry[1] = c2.entry[1]) ⟹ (c1 = c2)
        BY DEF StateMachineSafety
  ⟨1⟩2. TAKE c1, c2 ∈ committed
  ⟨1⟩3. SUFFICES ASSUME c1.entry[1] = c2.entry[1], c1 ≠ c2
        PROVE FALSE OBVIOUS
  ⟨1⟩4. c1.term ≠ c2.term
      ⟨2⟩1. SUFFICES ASSUME c1.term = c2.term
            PROVE FALSE OBVIOUS
      ⟨2⟩2. c1.entry[2] = c2.entry[2]
            BY ⟨2⟩1 DEF MRRInd,
            CommittedTermMatchesEntry
      ⟨2⟩3. c1.entry[1] = c2.entry[1] BY ⟨1⟩3
      ⟨2⟩4. c1 = c2
            BY ⟨1⟩0, ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, Z3 DEF TypeOK
      ⟨2⟩. QED BY ⟨1⟩3, ⟨2⟩4
  ⟨1⟩5. PICK s1 ∈ Server : InLog(c1.entry, s1)
        BY CommitsAreLogEntries
  ⟨1⟩6. PICK s2 ∈ Server : InLog(c2.entry, s2)
        BY CommitsAreLogEntries
  ⟨1⟩7. log[s1][c1.entry[1]] = c1.term
        BY ⟨1⟩5 DEF MRRInd, CommittedTermMatchesEntry,
        InLog, TypeOK
  ⟨1⟩8. log[s2][c2.entry[1]] = c2.term
        BY ⟨1⟩6 DEF MRRInd, CommittedTermMatchesEntry,
        InLog, TypeOK
  ⟨1⟩9. CASE c1.term > c2.term
      ⟨2⟩1. ∃ i ∈ DOMAIN log[s1] : log[s1][i] = c1.term
            BY ⟨1⟩5 DEF MRRInd,
            CommittedTermMatchesEntry, InLog, TypeOK
      ⟨2⟩2. ∃ i ∈ DOMAIN log[s1] : log[s1][i] > c2.term
            BY ⟨1⟩9, ⟨2⟩1 DEF TypeOK
      ⟨2⟩3. Len(log[s1]) ≥ c2.entry[1]
                ∧ log[s1][c2.entry[1]] = c2.term
          ⟨3⟩1. c2.term ≤ c2.term BY DEF MRRInd, TypeOK
          ⟨3⟩. QED BY ⟨1⟩5, ⟨2⟩2, ⟨3⟩1 DEF MRRInd, TypeOK,
            LogsLaterThanCommittedMustHaveCommitted
      ⟨2⟩4. log[s1][c1.entry[1]] = c2.term
            BY ⟨1⟩3, ⟨2⟩3 DEF MRRInd, TypeOK,
            CommittedEntryIndexesAreNonZero
      ⟨2⟩. QED BY ⟨1⟩4, ⟨1⟩7, ⟨2⟩4 DEF TypeOK
  ⟨1⟩10. CASE c1.term < c2.term
      ⟨2⟩1. ∃ i ∈ DOMAIN log[s2] : log[s2][i] = c2.term
            BY ⟨1⟩6 DEF MRRInd, InLog, TypeOK,
            CommittedTermMatchesEntry
      ⟨2⟩2. ∃ i ∈ DOMAIN log[s2] : log[s2][i] > c1.term
            BY ⟨1⟩10, ⟨2⟩1 DEF TypeOK
      ⟨2⟩3. Len(log[s2]) ≥ c1.entry[1]
                ∧ log[s2][c1.entry[1]] = c1.term
          ⟨3⟩1. c1.term ≤ c1.term BY DEF MRRInd, TypeOK
          ⟨3⟩. QED BY ⟨1⟩6, ⟨2⟩2, ⟨3⟩1 DEF MRRInd, TypeOK,
            LogsLaterThanCommittedMustHaveCommitted
      ⟨2⟩4. log[s2][c2.entry[1]] = c1.term
            BY ⟨1⟩3, ⟨2⟩3 DEF MRRInd, TypeOK,
            CommittedEntryIndexesAreNonZero
      ⟨2⟩. QED BY ⟨1⟩4, ⟨1⟩8, ⟨2⟩4 DEF TypeOK
  ⟨1⟩. QED BY ⟨1⟩4, ⟨1⟩9, ⟨1⟩10 DEF MRRInd, TypeOK
```

Figure 5. The TLAPS proof of Lemma 5.3.

The Raft protocol, upon initial publication, included a TLA+ formal specification of its static protocol, without dynamic reconfiguration [27]. Later, a formal verification of the safety properties of the static Raft protocol was completed using the Verdi framework for distributed systems verification [44]. The formal verification of static Raft in Verdi consisted of approximately 50,000 lines of Coq [3], took around 18 months to develop, and consisted of 90 total invariants. In comparison, our proof consists of 3189 lines of TLA+ code. Note, however, that it is difficult to directly compare our work with [44] because (1) our TLA+ specifications are written at a higher level of abstraction, and (2) part of the work in [44] was aimed at producing a verified, runnable Raft implementation, which was not our goal. The work of [44] did not include a verification of Raft's dynamic reconfiguration protocols. To our knowledge, our work is the first formally verified safety proof for a reconfiguration protocol that integrates with a Raft based system.

In general, developing formally verified proofs and inductive invariants for real world distributed protocols remains a challenging and non-trivial problem. In recent years, tools like Ivy [31] have attempted to ease the burden of inductive invariant discovery and verification by taking an interactive approach to invariant development, and constraining the specification language for describing these systems so it falls into a decidable fragment of first order logic [32]. These restrictions, however, can place additional burden on the user in cases where a protocol or its invariants do not naturally fall into this decidable fragment [30].

Recent work has built on top of the Ivy system in an attempt to automatically infer inductive invariants for distributed protocols, with varying degrees of success. Tools like IC3PO [12, 13], SWISS [14], and DistAI [45] represent the state of the art in automated inductive invariant discovery for distributed protocols. With some human guidance, they have recently been able to scale to larger protocols like Paxos, but have not yet been applied to protocols like Raft.

Apalache [17] is a symbolic model checker for TLA+ specifications that has been developed in recent years and can check inductiveness of protocol invariants for bounded parameters. It does not, however, currently have any procedures for automatic discovery of inductive invariants. In future it would be interesting to compare the effectiveness of using Apalache versus the probabilistic, TLC-based method for finding counterexamples to induction when debugging a candidate inductive invariant.

7 Conclusions and Future Work

In this paper we presented, to our knowledge, the first formal verification of a reconfiguration protocol for a Raft based replication system. We used TLA+ and TLAPS, the TLA+ proof system, to formalize and mechanically verify our inductive invariant and safety proofs.

In future, we are interested in exploring ways to further automate the inductive invariant discovery process to the extent possible. Formal verification of liveness properties of *MongoRaftReconfig* is another possible avenue for future efforts. In addition, we are interested in examining how the compositional structure of the protocol could be exploited to improve the inductive invariant discovery or TLAPS proof process.

References

[1] 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).

[2] Noran Azmy, Stephan Merz, and Christoph Weidenbach. 2016. A Rigorous Correctness Proof for Pastry. In *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016) (LNCS, Vol. 9675)*, Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklós Biró (Eds.). Springer, 86–101.

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

[4] Richard Bonichon, David Delahaye, and Damien Doligez. 2007. Zenon: An extensible automated theorem prover producing checkable proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 151–165.

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. *Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[6] 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.

[7] 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*. 398–407.

[8] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[9] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. 2012. TLA+ Proofs. *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science* 7436 (January 2012), 147–154. https://www.microsoft.com/en-us/research/publication/tla-proofs/

[10] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme Modelling in Practice. *Proc. VLDB Endow.* 13, 9 (may 2020), 1346–1358. https://doi.org/10.14778/3397230.3397233

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Aman Goel and Karem Sakallah. 2021. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods Symposium*. Springer, 131–150.

[13] Aman Goel and Karem A. Sakallah. 2021. Towards an Automatic Proof of Lamport's Paxos. In *Formal Methods in Computer-Aided Design (FMCAD)*, Ruzica Piskac and Michael W Whalen (Eds.). New Haven, Connecticut, 112–122. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_20

[14] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 115–131. https://www.usenix.org/conference/nsdi21/presentation/hance

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 1–17.

[16] 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

[17] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 123 (oct 2019), 30 pages. https://doi.org/10.1145/3360549

[18] Leslie Lamport. 1995. How to write a proof. *The American mathematical monthly* 102, 7 (1995), 600–608.

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

[20] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

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

[22] Leslie Lamport. 2018. Using TLC to Check Inductive Invariance. https://lamport.azurewebsites.net/tla/inductive-invariant.pdf

[23] Leslie Lamport and Lawrence C Paulson. 1999. Should your specification language be typed. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 502–526.

[24] Zohar Manna and Amir Pnueli. 2012. *Temporal verification of reactive systems: safety*. Springer Science & Business Media.

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

[26] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 358–372.

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

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

[29] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[30] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (oct 2017), 31 pages. https://doi.org/10.1145/3140568

[31] 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* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

[32] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. 2010. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning* 44, 4 (2010), 401–424.

[33] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

[34] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 43–57. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports

[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, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2071–2081. https://doi.org/10.14778/3352063.3352125

[37] William Schultz and Ian Dardik. 2021. *TLAPS Safety Proof of MongoRaftReconfig*. https://doi.org/10.5281/zenodo.5768484

[38] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. 2022. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

[39] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. 2012. Dynamic Reconfiguration of Primary/Backup Clusters. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 425–437. https://www.usenix.org/conference/atc12/technical-sessions/presentation/shraer

[40] Pierre Sutra. 2020. On the correctness of Egalitarian Paxos. *Inform. Process. Lett.* 156 (2020), 105901.

[41] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[42] Marko Vukolić et al. 2013. The origin of quorum systems. *Bulletin of EATCS* 2, 101 (2013).

[43] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. 2008. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 33–38.

[44] 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 *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

[45] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

[46] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–66.

[47] 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, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou