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, Leader-Completeness 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

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) Leader-Completeness, 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 theMongoRaftReconfig 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 thatMongoRaftReconfig 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 forMongoRaftReconfig, and Section 5 presents our formal safety proof ofMongoRaftReconfig 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, asMongoStaticRaft, to distinguish it from theMongoRaftReconfig 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, sinceMongoRaftReconfig is built on top ofMongoStaticRaft.

A MongoDB replica set runningMongoRaftReconfig consists of a set of server processes,Server = {s1, s2, . . . , sn}. 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 ofMongoStaticRaft 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 ofMongoRaftReconfig 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 ∈ 2Server is a subset of all servers, v ∈ N is a numeric configuration version, and t ∈ 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 ofMongoRaftReconfig, 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 ofMongoRaftReconfig. 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]. SinceMongoRaftReconfig is formally specified using the TLA+ language and it is the
2.3.1 Specifications in TLA+. Specifying a system in TLA+ consists of defining a set of state variables, \( \text{vars} \), 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, \( \text{Init} \), and a next state relation, \( \text{Next} \), 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 \( \text{Init} \land \Box \lbrack \text{Next} \rbrack_{\text{vars}} \), where \( \Box \) denotes the “always” operator of temporal logic, meaning that a formula holds true at every step of a behavior, and \( \text{vars} \) denotes a sequence of all state variables of a specification. \( \lbrack \text{Next} \rbrack_{\text{vars}} \) is equivalent to the expression \( \text{Next} \lor (\text{vars}' = \text{vars}) \), 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 \( A_1 \lor A_2 \lor \ldots \lor A_n \) 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 \( P \) if 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 \textsc{assume-prove} idiom treats the lemma as an implication. That is, if \textit{Conditions} hold, then \textit{Implication} must follow. Leaf statements are proved using the \textit{by} statement, and can reference theorems and lemmas by name, operator definitions, and previous statements by label. Each structural proof must end with a \textsc{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 presented in this paper, so we give a brief overview of the specification here. The complete specification can be found in the \texttt{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 \texttt{MongoRaftReconfig} are summarized in Figure 3. The operator \( \text{Quorums}(m) \) is defined as the set of all majority quorums [42] for a given set of servers \( m \). Reconfigurations are modeled by the \texttt{Reconfig}(s, m) action, which represents a reconfiguration that occurs on primary server \( s \) to a new configuration with member set \( m \in 2^\text{Server} \). Configuration propagation is modeled by the \texttt{SendConfig}(s, t) action, which represents the propagation of a configuration from server \( s \) to server \( t \). Elections are modeled by the action \texttt{BecomeLeader}(s, Q), which represents the election of server \( s \) by a set of voters \( Q \). The action \texttt{UpdateTerms}(s, t) propagates the term of a server \( s \) to server \( t \), if the term of \( s \) is newer than \( t \). The actions \texttt{ClientRequest}(s), \texttt{GetEntries}(s, t), \texttt{RollbackEntries}(s, t), and \texttt{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

![Figure 1. Example of a hierarchically structured TLAPS proof.](image1)

![Figure 2. The state variables of the MongoRaftReconfig protocol and their corresponding types stated as a type correctness predicate in TLA+. The notation \([A \rightarrow B]\) represents the set of all functions from set \( A \) to set \( B \) and \( \text{Seq}(S) \) represents the set of all sequences containing elements from the set \( S \).](image2)

![Figure 3. The state variables of the \texttt{MongoRaftReconfig} protocol and their corresponding types stated as a type correctness predicate in TLA+.](image3)
MRRInit ≜ MRRNext ≜ MRRSpec

which are two key, high level safety properties of both the
MongoDB replication system and standard Raft. Informally,
protocol satisfies LeaderCompleteness and StateMachineSafety.
In this paper we establish that the
3 Verification Problem Statement

MRRSpec ≜ MRRInit ∧ □[MRRNext]vars

Figure 3. Summary of the MongoRaftReconfig TLA+ speci-

fication. 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 character-
istics 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 veri-
yfing 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 pro-
tocol 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 Defini-
tion 3.1, where committed ∈ ℕ × ℕ refers to the set of com-
mitted log entries as (index, term) pairs, and InLog(i, t, s)
is a predicate determining whether a log entry (i, t) is con-
tained 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 Defini-
tion 3.2.

Definition 3.1 (Leader Completeness).
∀s ∈ Server : ∀(cindex, cterm) ∈ committed :
(state[s] = Primary ∧ cterm < term[s]) ⇒
InLog(cindex, cterm, s)

Definition 3.2 (State Machine Safety).
∀(ind1, t1, ind2, t2) ∈ committed :
(ind1 = ind2) ⇒ (t1 = t2)

Both LeaderCompleteness and StateMachineSafety are safety
properties. More specifically, they are both invariants, mean-
ing that they must hold in all reachable states of Mongo-
RaftReconfig. Thus, our verification goals can be stated for-
tomally 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 ⇒ □LeaderCompleteness

Theorem 3.4 (MRRImpliesStateMachineSafety).
MRRSpec ⇒ □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 ma-
terial [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 ⇒ □Inv

(1)
Suppose that Spec is of the form Spec = Init ∧ □[Next]vars, as
in the case of MongoRaftReconfig. Then, in order to estab-
lish Formula 1, it is sufficient to find a state predicate Ind
The inductive invariant that we developed for MongoDBReconfig is referred to as \( \text{MRRInd} \) and consists of 20 high level conjuncts, shown in Figure 4. The initial set of 6 conjuncts, labeled as \( E_1 \) in Figure 4, along with \( \text{TypeOK} \), 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.

![Figure 4. Our inductive invariant for MongoDBReconfig.](image)

such that the following conditions hold:

\[
\text{Init} \Rightarrow \text{Ind} \quad (2) \\
\text{Ind} \land \text{Next} \Rightarrow \text{Ind}' \quad (3) \\
\text{Ind} \Rightarrow \text{Inv} \quad (4)
\]

Conditions 2 and 3 are referred to as \textit{initiation} and \textit{conclusion}, respectively, and they are sufficient to show that \text{Ind} is an inductive invariant. Conditions 2, 3, and 4 are together sufficient to establish Formula 1.

In our case, \text{Spec} instantiates to \text{MRRSpec} and we have two instances of \text{Inv}, namely, LeaderCompleteness andStateMachineSafety. 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 MongoDBReconfig is referred to as \( \text{MRRInd} \) 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 \text{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, \( \text{TypeOK} \), establishes basic type-correctness constraints on the state variables of MongoDBReconfig. This is necessary in most cases when stating inductive invariants in TLA+, since it is an untyped formalism [23]. The full definition of \( \text{TypeOK} \) is shown in Figure 2. The initial set of 6 conjuncts, labeled as \( E_1 \) in Figure 4, along with \( \text{TypeOK} \), 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 \( \text{MRRInd} \) 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 \text{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 \( \text{MRRNext} \), where \( s \) satisfies \text{Inv} and \( t \) violates \text{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).

\[
\begin{align*}
\text{MRRInit} & \Rightarrow \text{MRRInd} \quad \text{(a)} \\
\text{MRRInd} \land \text{MRRNext} & \Rightarrow \text{MRRInd}' \quad \text{(b)}
\end{align*}
\]

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):

\[
\text{MRRInd} \triangleq I_1 \land I_2 \land \ldots \land I_{20}
\]

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

\[
\begin{align*}
\text{MRRInd} & \land \text{MRRNext} \Rightarrow I_1' \\
\text{MRRInd} & \land \text{MRRNext} \Rightarrow I_2' \\
& \vdots \\
\text{MRRInd} & \land \text{MRRNext} \Rightarrow I_{20}'
\end{align*}
\]

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

\[
\text{MRRNext} \triangleq A_1 \lor A_2 \lor \cdots \lor A_8
\]

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

\[
\begin{align*}
\text{MRRInd} & \land A_1 \Rightarrow I_j' \\
\text{MRRInd} & \land A_2 \Rightarrow I_j' \\
& \vdots \\
\text{MRRInd} & \land A_8 \Rightarrow I_j'
\end{align*}
\]

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 \(8 \times 20 = 160\) 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 LeaderCompleteness 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.

\[
\text{MRRInd} \Rightarrow \text{LeaderCompleteness}
\]

Lemma 5.3 (IndImpliesStateMachineSafety).

\[
\text{MRRInd} \Rightarrow \text{StateMachineSafety}
\]
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).

\[ \text{MRRInd} \Rightarrow \forall c \in \text{committed} : \exists s \in \text{Server} : \]
\[ \text{InLog}(c.\text{entry}, s) \]

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 (1).1. Steps (1)2 and (1)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. (1)4 is a composite proof that shows that c1 and c2 cannot share the same term. Steps (1)5 through (1)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 (1)9 and (1)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 (1)9 and (1)10 because of step (1)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 are 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 \( \mathbb{N} \) (NaturalInduction.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].
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 inductive properties 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


