Formal Verification of Multi-Paxos for Distributed Consensus[⋆]

Saksham Chand, Yanhong A. Liu, and Scott D. Stoller
Computer Science Department, Stony Brook University, Stony Brook, NY 11794

Abstract. This paper describes formal specification and verification of Lamport's Multi-Paxos algorithm for distributed consensus. The specification is written in TLA+, Lamport's Temporal Logic of Actions. The proof is written and checked using TLAPS, a proof system for TLA+. Building on Lamport, Merz, and Doligez's specification and proof for Basic Paxos, we aim to facilitate the understanding of Multi-Paxos and its proof by minimizing the difference from those for Basic Paxos, and to demonstrate a general way of proving other variants of Paxos and other sophisticated distributed algorithms. We also discuss our general strategies for proving properties about sets and tuples that helped the proof check succeed in significantly reduced time.

1 Introduction

Distributed consensus is a fundamental problem in distributed computing. It requires that a set of processes agree on some value or values. Consensus is essential when distributed services are replicated for fault-tolerance, because non-faulty replicas must agree. Unfortunately, consensus is difficult when the processes or communication channels may fail.

Paxos [17] is an important algorithm, developed by Lamport, for solving distributed consensus. Basic Paxos is for agreeing on a one-shot value, such as whether to commit a database transaction. Multi-Paxos is for agreeing on an infinite sequence of values, for example, a stream of commands to execute. Multi-Paxos has been used in many important distributed services, e.g., Google's Chubby [1,2] and Microsoft's Autopilot [13]. There are other Paxos variants, e.g., that reduce a message delay [20] or add preemption [18], but Multi-Paxos is the most important in making Paxos practical for distributed services that must perform a continual sequence of operations.

Paxos handles processes that run concurrently without shared memory, where processes may crash and may later recover, and messages may be delayed indefinitely or lost. In Basic Paxos, each process may repeatedly attempt to be the leader and propose some value, and wait for appropriate replies from appropriate subsets of the processes while also replying appropriately to other processes; consensus is reached eventually if enough processes and channels are non-faulty

[⋆] This work was supported in part by NSF grants CCF-1414078, CCF-1248184, and CNS-1421893, ONR grant N000141512208, and AFOSR grant FA9550-14-1-0261. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

to elect a leader. In Multi-Paxos, many more different attempts, proposals, and replies may happen in overlapping fashions to reach consensus on values in different slots in the continual sequence.

Paxos has often been difficult to understand since it was created in the late 1980s [22]. Lamport later wrote a much simpler description of the phases of the algorithm but only for Basic Paxos [18]. Lamport, Merz, and Doligez [23] wrote a formal specification and proof of Basic Paxos in TLA+ [19] and TLAPS [27]. Many efforts, especially in recent years, have been spent on formal specification and verification of Multi-Paxos, but they use more restricted or less direct language models, some mixed in large systems with many unrelated functionalities, or handle other variants of Paxos than Multi-Paxos, as discussed in Section 7. What is lacking is formal specification and proof of the exact phases of Multi-Paxos, in a most direct and general language like TLA+ [19], with a complete proof that is mechanically checked, and a general method for doing such specifications and proofs in a more feasible way.

This paper addresses this challenge. We describe a formal specification of Multi-Paxos written in TLA+, and a complete proof written and automatically checked using TLAPS. Building on Lamport, Merz, and Doligez's specification and proof for Basic Paxos, we aim to facilitate the understanding of multi-Paxos and its proof by minimizing the difference from those for Basic Paxos. The key change in the specification is to replace operations involving two numbers with those involving a set of 3-tuples, for each of a set of processes, exactly capturing the minimum conceptual difference between Basic Paxos and Multi-Paxos. However, the proof becomes significantly more difficult because of the handling of sets and tuples in place of two numbers.

This work also aims to show the minimum-change approach as a general way of specifying and verifying other variants of Paxos, and more generally of specifying and verifying other sophisticated algorithms by starting from the basics. We demonstrate this by further showing the extension of the specification and proof of Multi-Paxos to add preemption—letting processes abandon proposals that are already preempted by other proposals [18,30]. We also extended the specification and proof of Basic Paxos with preemption, which is even easier.

Finally, we discuss a general method we attempted to follow to tackle tedious and difficult proof obligations involving sets and tuples, a well-known significant complication in general. For difficult properties involving sets, we use induction and direct the prover to focus on the changes in the set values. For properties involving tuples, we change the ways of accessing and testing the elements to yield significantly reduced proof-checking time. Overall, we were able to keep the specification minimumly changed, and keep the proof-checking time to about 2 minutes or less while the prover checks the proofs for over 900 obligations for both Multi-Paoxs and Multi-Paxos with Preemption.

Our full TLA+ specification and TLAPS-checked proof of Multi-Paxos with Preemption are included in Appendix B.

2 Distributed consensus and Paxos

A system is a set of processes that can process values individually and can communicate with each other by sending and receiving messages. The processes may crash and may later recover, and the messages may be delayed indefinitely or lost.

Distributed consensus. The basic consensus problem, called single-value consensus or single-decree consensus, is to ensure that a single value is chosen from among the values proposed by the processes. The safety requirements for basic consensus are [18]:

- Only a value that has been proposed may be chosen.
- Only a single value is chosen.
- A process never learns that a value has been chosen unless it actually has been chosen.

Formally this is defined as

$$\mathit{Consistency}_{basic} \quad \triangleq \quad \forall\, v_1, v_2 \in \mathcal{V} \,:\, \phi(v_1) \wedge \phi(v_2) \Rightarrow v_1 = v_2 \tag{1}$$

where $\mathcal{V}$ is the set of possible proposed values, and ϕ is a predicate that given a value v evaluates to true iff v was chosen by the algorithm. The specification of ϕ is part of the algorithm.

The more general consensus problem, called multi-value consensus or multi-decree consensus, is to choose a sequence of values, instead of a single value. Here we have

$$\mathit{Consistency}_{multi} \quad \triangleq \quad \forall\, v_1, v_2 \in \mathcal{V}, s \in \mathcal{S} \,:\, \phi(v_1, s) \wedge \phi(v_2, s) \Rightarrow v_1 = v_2 \tag{2}$$

where $\mathcal{V}$ is as above, $\mathcal{S}$ is a set of *slots* used to index the sequence of decisions, and ϕ is a predicate that given a value v and a slot s evaluates to true iff v was chosen for s by the algorithm.

Basic Paxos and Multi-Paxos. Paxos solves the problem of consensus. Two main roles of the algorithm are performed by two kinds of processes:

- $\mathcal{P}$ is the set of proposers. These processes propose values that can be chosen.
- $\mathcal{A}$ is the set of acceptors. These processes vote for proposed values. A value is chosen when there are enough votes for it.

A set $\mathcal{Q}$ of subsets of the acceptors, i.e., $\mathcal{Q} \subseteq 2^{\mathcal{A}}$, is used as a quorum system. It must satisfy the following properties:

- $\mathcal{Q}$ is a set cover for $\mathcal{A}$, i.e., $\bigcup_{Q \in \mathcal{Q}} Q = \mathcal{A}$.
- Any two quorums overlap, i.e., $\forall Q_1, Q_2 \in \mathcal{Q} \,:\, Q_1 \cap Q_2 \neq \emptyset$.

The most commonly used quorum system $\mathcal{Q}$ takes any majority of acceptors as an element in $\mathcal{Q}$.

Basic Paxos solves the problem of single-value consensus. It defines predicate ϕ as

$$\phi(v) \;\triangleq\; \exists Q \in \mathcal{Q} : \forall a \in Q : \exists b \in \mathcal{B} : \mathit{sent}(\text{“}\mathsf{2b}\text{”}, b, v, a) \tag{3}$$

where $\mathcal{B}$ is the set of proposal numbers, also called ballot numbers, which is any set that can be strictly totally ordered. $\mathit{sent}(\text{“}\mathsf{2b}\text{”}, b, v, a)$ means that a message of type “2b” with ballot number b and value v was sent by acceptor a (to some set of processes). An acceptor votes by sending such a message.

Multi-Paxos solves the problem of multi-value consensus. It trivially extends predicate ϕ to decide a value for each slot s in $\mathcal{S}$:

$$\phi(v, s) \;\triangleq\; \exists Q \in \mathcal{Q} : \forall a \in Q : \exists b \in \mathcal{B} : \mathit{sent}(\text{“}\mathsf{2b}\text{”}, b, v, a, s) \tag{4}$$

To satisfy the safety requirements, $\mathcal{S}$ need not have any relations defined on it. In practice, $\mathcal{S}$ is usually the natural numbers.

> Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.
>
> **Phase 1.** (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.
> (b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.
>
> **Phase 2.** (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
> (b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.
>
> A proposer can make multiple proposals, so long as it follows the algorithm for each one. ... It is probably a good idea to abandon a proposal if some proposer has begun trying to issue a high-numbered one. Therefore, if an acceptor ignores a *prepare* or *accept* request because it has already received a *prepare* request with a higher number, then it should probably inform the propose, who should then abandon its proposal. This is a performance optimization that does not affect correctness.
>
> ---
>
> To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. ...

Fig. 1. Lamport's description of Basic Paxos in English [18].

Figure 1 shows Lamport's description of Basic Paxos [18]. It uses any majority of acceptors as a quorum. Also, in Phase 2a, it instructs the *accept* request be sent to each of those acceptors that replied with the proposer's ballot number n, but it is sufficient for safety to send the *accept* request to any subset of $\mathcal{A}$. For liveness, however, the set of receivers should contain at least one quorum, which again is allowed to be different from the quorum that responded to n.

Multi-Paxos can be built from Basic Paxos by carefully adding slots. In Basic Paxos, acceptors cache the value they have accepted with the highest ballot number. With slots, we have a sequence of these values indexed by slot. Therefore,

- In Phase 1b, the acceptor now replies with a mapping in $\mathcal{S} \to \mathcal{B} \times \mathcal{V}$ as opposed to just one pair in $\mathcal{B} \times \mathcal{V}$.
- The same change is needed in Phase 2b.
- Upon receiving such a mapping as a reply, in Phase 2a, a proposer proposes a mapping in $\mathcal{S} \to \mathcal{V}$ instead of just one value in $\mathcal{V}$. In the same way that v was chosen in Basic Paxos, by picking the value backed by the highest received ballot number, in Multi-Paxos, the proposer does this calculation for each slot in the received mapping.
- Phase 1a is unchanged.
- Learning, as described in the last part of Figure 1, is also unchanged, except to consider different slots separately—a process learns that a value is chosen for a slot if a quorum of acceptors accepted it for that slot.

3 Specification of Multi-Paxos

We give a formal specification of Multi-Paxos by minimally extending that of Basic Paxos by Lamport, Merz, and Doligez [23].

Variables. The specification of Multi-Paxos has four global variables.

msgs—the set of messages that have been sent. Processes read from or add to this set. This is the same as in the specification of Basic Paxos.

accVoted—per acceptor, a set of triples in $\mathcal{B} \times \mathcal{S} \times \mathcal{V}$, capturing a mapping in $\mathcal{S} \to \mathcal{B} \times \mathcal{V}$, that the acceptor has voted for. This contrasts two numbers per acceptor, in two variables, *maxVBal* and *maxVal*, in the specification of Basic Paxos

accMaxBal—per acceptor, the highest ballot number seen by the acceptor. This is named *maxBal* in the specification of Basic Paxos.

proBallot—per proposer, the ballot number of the current ballot being run by the proposer. This is not in the specification of Basic Paxos; it is added to support preemption.

Algorithm steps. The algorithm consists of repeatedly executing two phases.

Phase 1a. Figure 2 shows the specifications of Phase 1a for Basic Paxos and Multi-Paxos, which are in essence the same. Parameter ballot number b in Basic Paxos is replaced with proposer p executing this phase in Multi-Paxos,

Basic Paxos	Multi-Paxos
$\mathit{Phase1a}(b \in \mathcal{B}) \;\triangleq$	$\mathit{Phase1a}(p \in \mathcal{P}) \;\triangleq$
$\wedge \nexists\, m \in \mathit{msgs} : (m.\mathit{type} = \text{“}\mathsf{1a}\text{”}) \wedge$	$\wedge \nexists\, m \in \mathit{msgs} : (m.\mathit{type} = \text{“}\mathsf{1a}\text{”}) \wedge$
$(m.\mathit{bal} = b)$	$(m.\mathit{bal} = \mathit{proBallot}[p])$
$\wedge \mathit{Send}([\mathit{type} \mapsto \text{“}\mathsf{1a}\text{”},$	$\wedge \mathit{Send}([\mathit{type} \mapsto \text{“}\mathsf{1a}\text{”},$
$\mathit{bal} \mapsto b)$	$\mathit{bal} \mapsto \mathit{proBallot}[p], \mathit{from} \mapsto p])$
$\wedge \textsc{unchanged}\ \langle \mathit{maxVBal}, \mathit{maxBal},$	$\wedge \textsc{unchanged}\ \langle \mathit{accVoted}, \mathit{accMaxBal},$
$\mathit{maxVal} \rangle$	$\mathit{proBallot} \rangle$

Fig. 2. Phase 1a of Basic Paxos and Multi-Paxos

Basic Paxos	Multi-Paxos
$\mathit{Phase1b}(a \in \mathcal{A}) \;\triangleq$	$\mathit{Phase1b}(a \in \mathcal{A}) \;\triangleq$
$\exists\, m \in \mathit{msgs} :$	$\exists\, m \in \mathit{msgs} :$
$\wedge m.\mathit{type} = \text{“}\mathsf{1a}\text{”}$	$\wedge m.\mathit{type} = \text{“}\mathsf{1a}\text{”}$
$\wedge m.\mathit{bal} > \mathit{maxBal}[a]$	$\wedge m.\mathit{bal} > \mathit{accMaxBal}[a]$
$\wedge \mathit{Send}([\mathit{type} \mapsto \text{“}\mathsf{1b}\text{”},$	$\wedge \mathit{Send}([\mathit{type} \mapsto \text{“}\mathsf{1b}\text{”},$
$\mathit{bal} \mapsto m.\mathit{bal},$	$\mathit{bal} \mapsto m.\mathit{bal},$
$\mathit{maxVBal} \mapsto \mathit{maxVBal}[a],$	$\mathit{voted} \mapsto \mathit{accVoted}[a],$
$\mathit{maxVal} \mapsto \mathit{maxVal}[a],$	
$\mathit{acc} \mapsto a])$	$\mathit{from} \mapsto a])$
$\wedge \mathit{maxBal}' =$	$\wedge \mathit{accMaxBal}' =$
$[\mathit{maxBal}\ \textsc{except}\ ![a] = m.\mathit{bal}]$	$[\mathit{accMaxBal}\ \textsc{except}\ ![a] = m.\mathit{bal}]$
$\wedge \textsc{unchanged}\ \langle \mathit{maxVBal}, \mathit{maxVal} \rangle$	$\wedge \textsc{unchanged}\ \langle \mathit{accVoted}, \mathit{proBallot} \rangle$

Fig. 3. Phase 1b of Basic Paxos and Multi-Paxos

to allow extensions such as preemption that need to know the proposer of a ballot number; uses of b are changed to $\mathit{proBallot}[p]$; and $\mathit{from} \mapsto p$ is added in *Send*. *Send* is a macro that adds its argument to *msgs*, i.e., $\mathit{Send}(m) \;\triangleq\; \mathit{msgs}' = \mathit{msgs} \cup \{m\}$. In this specification, 1a messages do not have a receiver, making them accessible to all processes. However, this is not required. For safety, it is enough to send this message to *any* subset of $\mathcal{A}$, even $\emptyset$. For liveness, the receiving set should contain at least one quorum.

Phase 1b. Figure 3 shows the specifications of Phase 1b. Parameter acceptor a executes this phase. The only key difference between the specifications is the set $\mathit{accVoted}[a]$ of triples in *Send* of Multi-Paxos vs. the two numbers $\mathit{maxVBal}[a]$ and $\mathit{maxVal}[a]$ in Basic Paxos.

Phase 2a. Figure 4 shows Phase 2a. The key difference is, in *Send*, the bloating of a single value v in $\mathcal{V}$ in Basic Paxos to a set of pairs given by *ProposeDecrees* capturing a mapping in $\mathcal{S} \to \mathcal{V}$ in Multi-Paxos. The operation of finding the value with the highest ballot in Basic Paxos is performed for each slot by *Bmax* in Multi-Paxos; *Bmax* takes a set T of triples capturing a mapping in $\mathcal{S} \to \mathcal{B} \times \mathcal{V}$ and returns a set of pairs capturing a mapping in $\mathcal{S} \to \mathcal{V}$. *NewProposals* generates a set of pairs capturing a mapping in $\mathcal{S} \to \mathcal{V}$ where values are proposed for slots not in *Bmax*. Note that this is significantly more sophisticated than running Basic Paxos for each slot, because the ballots are shared and changing for all slots, and slots are paired with values dynamically where slots that failed to reach consensus values earlier are also detected and reused.

Basic Paxos	Multi-Paxos
$Phase2a(b \in \mathcal{B}) \ \triangleq$ $\land \nexists m \in msgs : (m.type = \text{“2a”}) \land$ $(m.bal = b)$ $\land \exists\, v \in \mathcal{V} :$ $\land \exists\, Q \in \mathcal{Q} : \exists\, S \in \text{SUBSET}\ \{m \in msgs :$ $(m.type = \text{“1b”}) \land$ $(m.bal = b)\} :$ $\land \forall\, a \in Q : \exists\, m \in S : m.acc = a$ $\land \lor \forall\, m \in S : m.maxVBal = -1$ $\lor \exists\, c \in 0..(b-1) :$ $\land \forall\, m \in S : m.maxVBal =< c$ $\land \exists\, m \in S : (m.maxVBal = c)$ $\land m.maxVal = v$ $\land Send([type \mapsto \text{“2a”}, bal \mapsto b, val \mapsto v])$ $\land \text{UNCHANGED}\ \langle maxBal, maxVBal,$ $maxVal \rangle$	$Phase2a(p \in \mathcal{P}) \ \triangleq$ $\land \nexists m \in msgs : (m.type = \text{“2a”}) \land$ $(m.bal = proBallot[p])$ $\land \exists\, Q \in \mathcal{Q}, S \in \text{SUBSET}\ \{m \in msgs :$ $(m.type = \text{“1b”}) \land$ $(m.bal = proBallot[p])\} :$ $\land \forall\, a \in Q : \exists\, m \in S : m.from = a$ $\land Send([type \mapsto \text{“2a”},$ $bal \mapsto proBallot[p],$ $decrees \mapsto ProposeDecrees(\text{UNION}$ $\{m.voted : m \in S\}),$ $from \mapsto p])$ $\land \text{UNCHANGED}\ \langle accMaxBal, accVoted,$ $proBallot \rangle$ $Bmax(T) \ \triangleq$ $\{[slot \mapsto t.slot, val \mapsto t.val] : t \in$ $\{t \in T : \forall\, t2 \in T : t2.slot = t.slot$ $\Rightarrow t2.bal =< t.bal\}\}$ $FreeSlots(T) \ \triangleq$ $\{s \in \mathcal{S} : \nexists t \in T : t.slot = s\}$ $NewProposals(T) \ \triangleq$ $\text{CHOOSE}\ D \in (\text{SUBSET}\ [slot :$ $FreeSlots(T), val : \mathcal{V}]) \setminus \{\} :$ $\forall\, d1, d2 \in D : d1.slot = d2.slot \Rightarrow$ $d1 = d2$ $ProposeDecrees(T) \ \triangleq$ $Bmax(T) \cup NewProposals(T)$

Fig. 4. Phase 2a of Basic Paxos and Multi-Paxos

Phase 2b. Figure 5 shows Phase 2b. In Basic Paxos, the acceptor updates its voted pair $maxVBal[a]$ and $maxVal[a]$ upon receipt of a 2a message of the highest ballot. In Multi-Paxos, this is performed for each slot. The acceptor updates $accVoted$ to have all decrees in the received 2a message and all previous values in $accVoted$ for slots not mentioned in that message.

Complete algorithm specification. To complete the algorithm specification, we define *vars*, *Init*, *Next*, and *Spec*, typical TLA+ macro names for the set of

Basic Paxos	Multi-Paxos
$Phase2b(a \in \mathcal{A}) \triangleq$	$Phase2b(a \in \mathcal{A}) \triangleq$
$\exists\, m \in msgs :$	$\exists\, m \in msgs :$
$\wedge m.type = \text{“2a”}$	$\wedge m.type = \text{“2a”}$
$\wedge m.bal >= maxBal[a]$	$\wedge m.bal >= accMaxBal[a]$
$\wedge Send([type \mapsto \text{“2b”},$	$\wedge Send([type \mapsto \text{“2b”},$
$bal \mapsto m.bal,$	$bal \mapsto m.bal,$
$val \mapsto m.val,$	$decrees \mapsto m.decrees,$
$acc \mapsto a])$	$from \mapsto a)$
$\wedge maxBal' =$	$\wedge accMaxBal' =$
$[maxBal \text{ EXCEPT } ![a] = m.bal]$	$[accMaxBal \text{ EXCEPT } ![a] = m.bal]$
$\wedge maxVBal' =$	$\wedge accVoted' = [accVoted \text{ EXCEPT } ![a] =$
$[maxBal \text{ EXCEPT } ![a] = m.bal]$	$\cup\{[bal \mapsto m.bal, slot \mapsto d.slot,$
$\wedge maxVal' =$	$val \mapsto d.val] : d \in m.decrees\}]$
$[maxVal \text{ EXCEPT } ![a] = m.val]$	$\cup\{e \in accVoted[a] :$
	$\nexists\, r \in m.decrees : e.slot = r.slot\}$
	$\wedge \text{UNCHANGED } \langle proBallot \rangle$

Fig. 5. Phase 2b of Basic Paxos and Multi-Paxos

variables, the initial state, possible actions leading to the next state, and the system specification, respectively:

$$\begin{aligned}
vars &\triangleq \langle msgs, accVoted, accMaxBal, proBal \rangle \\
Init &\triangleq msgs = \{\} \wedge accVoted = [a \in \mathcal{A} \mapsto \{\}] \wedge \\
&\quad\;\; accMaxBal = [a \in \mathcal{A} \mapsto -1] \wedge proBal = [p \in \mathcal{P} \mapsto 0] \\
Next &\triangleq \vee \exists\, p \in \mathcal{P} : Phase1a(p) \vee Phase2a(p) \\
&\quad\;\; \vee \exists\, a \in \mathcal{A} : Phase1b(a) \vee Phase2b(a) \\
Spec &\triangleq Init \wedge \Box[Next]_{vars}
\end{aligned} \tag{5}$$

4 Verification of Multi-Paxos

We first define the auxiliary predicates and invariants used, by extending those for the proof of Basic Paxos with slots, and then describe our proof strategy which proves *Consistency* of Multi-Paxos.

Auxiliary predicates. These predicates are used throughout the proof. We define the predicate ϕ in (4) by $\phi(v, s) \equiv Chosen(v, s)$, where:

$$\begin{aligned}
&VotedForIn(a \in \mathcal{A}, v \in \mathcal{V}, b \in \mathcal{B}, s \in \mathcal{S}) \triangleq \\
&\quad \exists\, m \in msgs : \\
&\qquad m.type = \text{“2b”} \wedge m.bal = b \wedge m.from = a \wedge \\
&\qquad \exists\, d \in m.decrees : d.slot = s \wedge d.val = v \\
&ChosenIn(v \in \mathcal{V}, b \in \mathcal{B}, s \in \mathcal{S}) \triangleq \\
&\quad \exists\, Q \in \mathcal{Q} : \forall\, a \in Q : VotedForIn(a, v, b, s) \\
&Chosen(v \in \mathcal{V}, s \in \mathcal{S}) \triangleq \\
&\quad \exists\, b \in \mathcal{B} : ChosenIn(v, b, s)
\end{aligned} \tag{6}$$

Predicate $MaxVotedBallotInSlot(D \in \text{SUBSET}\,[slot : \mathcal{S}, bal : \mathcal{B}], s \in \mathcal{S})$ returns the highest ballot number among all pair in set D with slot s.

$$
\begin{aligned}
&Maximum(B) \triangleq \\
&\quad \text{CHOOSE } b \in B : \forall\, b2 \in B : b >= b2 \\
&MaxVotedBallotInSlot(D \in \text{SUBSET}\,[slot : \mathcal{S}, bal : \mathcal{B}], s \in \mathcal{S}) \triangleq \\
&\quad \text{LET } B \triangleq \{d.bal : d \in \{d \in D : d.slot = s\}\} \\
&\quad \text{IN} \quad \text{IF } \{d \in D : d.slot = s\} = \{\} \text{ THEN } -1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\; \text{ELSE } Maximum(B)
\end{aligned}
\tag{7}
$$

Type invariants. Type invariants are captured by $TypeOK$.

$$
\begin{aligned}
&Messages \triangleq \\
&\quad \cup\, [type : \{\text{“1a”}\}, bal : \mathcal{B}, from : \mathcal{P}] \\
&\quad \cup\, [type : \{\text{“1b”}\}, bal : \mathcal{B}, voted : \text{SUBSET}\,[bal : \mathcal{B}, slot : \mathcal{S}, val : \mathcal{V}], from : \mathcal{A}] \\
&\quad \cup\, [type : \{\text{“2a”}\}, bal : \mathcal{B}, decrees : \text{SUBSET}\,[slot : \mathcal{S}, val : \mathcal{V}], from : \mathcal{P}] \\
&\quad \cup\, [type : \{\text{“2b”}\}, bal : \mathcal{B}, from : \mathcal{A}, decrees : \text{SUBSET}\,[slot : \mathcal{S}, val : \mathcal{V}]] \\
&\quad \cup\, [type : \{\text{“preempt”}\}, bal : \mathcal{B}, to : \mathcal{P}, maxBal : \mathcal{B}] \\
&TypeOK \triangleq \\
&\quad \land\, msgs \in \text{SUBSET } Messages \\
&\quad \land\, accVoted \in [\mathcal{A} \rightarrow \text{SUBSET}\,[bal : \mathcal{B}, slot : \mathcal{S}, val : \mathcal{V}]] \\
&\quad \land\, accMaxBal \in [\mathcal{A} \rightarrow \mathcal{B} \cup \{-1\}] \\
&\quad \land\, proBallot \in [\mathcal{P} \rightarrow \mathcal{B}] \\
&\quad \land\, \forall\, a \in \mathcal{A} : \forall\, t \in accVoted[a] : accMaxBal[a] >= t.bal
\end{aligned}
\tag{8}
$$

Invariants about messages. The following invariant is for 1b messages. The first conjunct establishes that the ballot number is at most the highest ballot number seen by the sending acceptor. The second conjunct states that the decrees contained within the message body have been voted for by the sending acceptor. The last conjunct asserts that for each slot, relative to the timeline established by ballot numbers, since the last time this acceptor voted in the slot to the time this message was sent, no voting occurred on the slot by this acceptor.

$$
\begin{aligned}
&MsgInv1b \triangleq \\
&\quad \forall\, m \in msgs : (m.type = \text{“1b”}) \Rightarrow \\
&\qquad \land\, m.bal =< accMaxBal[m.from] \\
&\qquad \land\, \forall\, t \in m.voted : VotedForIn(m.from, t.val, t.bal, t.slot) \\
&\qquad \land\, \forall\, b2 \in \mathcal{B}, s \in \mathcal{S}, v \in \mathcal{V} : b2 \in (MaxVotedBallotInSlot(m.voted, s), m.bal) \\
&\qquad\quad \Rightarrow \neg\, VotedForIn(m.from, v, b2, s)
\end{aligned}
\tag{9}
$$

Proof strategy. The proof is developed following a standard hierarchical structure and uses proof by induction and contradiction.

$$\begin{array}{l}
\mathit{MsgInv} \;\triangleq\; \mathit{MsgInv1b} \land \mathit{MsgInv2a} \land \mathit{MsgInv2b} \\
\mathit{Inv} \;\triangleq\; \mathit{TypeOK} \land \mathit{AccInv} \land \mathit{MsgInv} \\
\mathit{Consistency} \;\triangleq\; \forall\, v_1, v_2 \in \mathcal{V}, s \in \mathcal{S} \;:\; \mathit{Chosen}(v_1, s) \land \mathit{Chosen}(v_2, s) \Rightarrow v_1 = v_2 \\
\textsc{theorem}\ \mathit{Consistent} \;\triangleq\; \mathit{Spec} \Rightarrow \Box \mathit{Consistency}
\end{array} \tag{10}$$

where *AccInv* is an invariant about acceptors, and $\mathit{MsgInv2a}$ and $\mathit{MsgInv2b}$ are invariants for 2a and 2b messages, respectively, and these three invariants are defined in Appendix A.

The main theorem to prove is *Consistent* as defined in Equation (10). For this, we define *Inv* and first prove $\mathit{Inv} \Rightarrow \mathit{Consistency}$. Then, we prove $\mathit{Spec} \Rightarrow \Box \mathit{Inv}$ which by temporal logic, concludes $\mathit{Spec} \Rightarrow \Box \mathit{Consistency}$. To prove $\mathit{Spec} \Rightarrow \Box \mathit{Inv}$, we employ a systematic proof strategy that works very well for algorithms described in the event driven paradigm, including message-passing distributed algorithms. We demonstrate the strategy for some invariants in *Inv*.

First, consider invariant *TypeOK*. The goal is $\mathit{Spec} \Rightarrow \Box \mathit{TypeOK}$. Recall $\mathit{Spec} \;\triangleq\; \mathit{Init} \land \Box[\mathit{Next}]_{vars}$. The induction basis, $\mathit{Init} \Rightarrow \mathit{TypeOK}$, is trivial, and TLAPS handles it automatically. Next, we want to prove $\mathit{TypeOK} \land [\mathit{Next}]_{vars} \Rightarrow \mathit{TypeOK}'$, where the left side is the induction hypothesis, and right side is the goal to be proved. $[\mathit{Next}]_{vars}$ is a disjunction of phases, as for any algorithm, and TypeOK' is a conjunction of smaller invariants, as for many invariants. Now, the basis can be stripped down to each disjunct separately, and each smaller goal needs to be proved from all smaller disjuncts. This process is mechanical, and TLAPS provides a feature for precisely this expansion into smaller proof obligations. This breakdown is the first step in our proof strategy. For *TypeOK*, this expands to 5 smaller assertions; with 5 phases in *Next*, we obtain 25 small proofs done by the prover with no manual intervention.

MsgInv and *AccInv* are more involved. We proceed like we did for *TypeOK* and create a proof tree, each branch of which aims to prove an invariant for some disjunct in *Next*. To explain the rest of our strategy, we show one combination: *MsgInv* and $\mathit{Phase1b}$. Equation (11) gives the skeleton of the proof; the full proof is in Appendix B. Goal for the prover is step $\langle 4 \rangle 2$ which states that MsgInv' holds if an acceptor, a, executes $\mathit{Phase1b}$. m is any message in the new set of messages, msgs'. Substeps $\langle 5 \rangle 1, 2, 3$ focus on $\mathit{MsgInv1b}$, $\mathit{MsgInv2a}$, $\mathit{MsgInv2b}$, respectively.

$\mathit{Phase1b}$ generates a $1b$ message. $\langle 5 \rangle 3$ is easy for the prover as it argues about $2b$ messages. Intuitively, $\langle 5 \rangle 2$ should be easy for the prover too since, like $\langle 5 \rangle 3$, it involves a message type that is not what $\mathit{Phase1b}$ generates. However, this is not the case because of predicate *SafeAt*, which is used in $\mathit{MsgInv2a}$ and expresses whether it is safe to accept a given value for a given ballot for a given slot (the formal definition is in Appendix A). At this point the prover needs a continuity lemma.

We define a *continuity lemma* as a lemma which asserts that a predicate continues to hold (or not hold) as the system goes from one state to the next in a single step. For example, the continuity lemma for *SafeAt* states that *SafeAt* continues to hold for any disjunct in *Next*, which includes $\mathit{Phase1b}(a)$. The characteristic property of such lemmas is their reuse. In our proof of Multi-Paxos, we defined 5 continuity lemmas which are asserted in 24 places.

Lastly, we need to prove $\langle 5\rangle 1$. Since $\langle 5\rangle 1$ asserts about $1b$ messages and $\mathit{Phase1b}$ generates such messages, the proof is more complicated and the prover needs manual intervention. Here we split the set of messages in the new state into two: $\langle 6\rangle 1$ for the old messages, and $\langle 6\rangle 2$ for the increment created in this step. For the old messages, we need continuity lemmas. The most challenging is the increment. To deal with the increment, we focus on the cause of the increment—the definition of $\mathit{Phase1b}$—and treat each goal conjunct separately in $\langle 7\rangle 1, 2, 3$. The prover proves $\langle 7\rangle 1$ by just the definition of $\mathit{Phase1b}$ and the fact that it is the increment. For $\langle 7\rangle 2$, along with the definition of $\mathit{Phase1b}$, the prover also needs continuity lemma for $\mathit{VotedForIn}$. $\langle 7\rangle 3$ required, along with the definition of $\mathit{Phase1b}$ and continuity lemmas, some problem-specific manual intervention. In this case, we helped the prover understand the change in limits of the set $\mathit{MaxVotedBallotInSlot}(m.\mathit{voted}, s) + 1..m.\mathit{bal} - 1$.

```
$\langle 4\rangle 2.$ASSUME NEW $a \in \mathcal{A}$, NEW $m \in \mathit{msgs}'$, $\mathit{Phase1b}(a)$ PROVE $\mathit{MsgInv}'$
  $\langle 5\rangle 1.((m.\mathit{type} = \text{“1b”}) \Rightarrow \ (*\ \mathit{MsgInv1b}'\ *)$
      $\land\ m.\mathit{bal} \leq \mathit{acceptorMaxBal}[m.\mathit{from}]$
      $\land\ \forall\, r \in m.\mathit{voted} : \mathit{VotedForIn}(m.\mathit{from}, r.\mathit{val}, r.\mathit{bal}, r.\mathit{slot})$
      $\land\ \forall\, s \in \mathcal{S}, v \in \mathcal{V}, c \in \mathcal{B} :$
          $c \in \mathit{MaxVotedBallotInSlot}(m.\mathit{voted}, s) + 1..m.\mathit{bal} - 1 \Rightarrow$
          $\neg\,\mathit{VotedForIn}(m.\mathit{from}, v, c, s))'$
          $\langle 6\rangle 1.$CASE $m \in \mathit{msgs}\ \ldots$
          $\langle 6\rangle 2.$CASE $m \in \mathit{msgs}' \setminus \mathit{msgs}$
              $\langle 7\rangle 1.(m.\mathit{bal} \leq \mathit{acceptorMaxBal}[m.\mathit{from}])'$
              $\langle 7\rangle 2.(\forall\, r \in m.\mathit{voted} : \mathit{VotedForIn}(m.\mathit{from}, r.\mathit{val}, r.\mathit{bal}, r.\mathit{slot}))'\ \ldots$      (11)
              $\langle 7\rangle 3.(\forall\, s \in \mathcal{S}, v \in \mathcal{V}, c \in \mathcal{B} :$
                  $c \in \mathit{MaxVotedBallotInSlot}(m.\mathit{voted}, s) + 1..m.\mathit{bal} - 1 \Rightarrow$
                  $\neg\,\mathit{VotedForIn}(m.\mathit{from}, v, c, s))'\ \ldots$
  $\langle 5\rangle 2.((m.\mathit{type} = \text{“2a”}) \Rightarrow \ (*\ \mathit{MsgInv2a}'\ *)$
      $\land\ \forall\, d \in m.\mathit{decrees} : \mathit{SafeAt}(d.\mathit{val}, m.\mathit{bal}, d.\mathit{slot})$
      $\land\ \forall\, d1, d2 \in m.\mathit{decrees} : d1.\mathit{slot} = d2.\mathit{slot} \Rightarrow d1 = d2$
      $\land\ \forall\, \mathit{ma} \in \mathit{msgs} : (\mathit{ma}.\mathit{type} = \text{“2a”}) \land (\mathit{ma}.\mathit{bal} = m.\mathit{bal}) \Rightarrow (\mathit{ma} = m))'\ \ldots$
  $\langle 5\rangle 3.((m.\mathit{type} = \text{“2b”}) \Rightarrow \ (*\ \mathit{MsgInv2b}'\ *)$
      $\land\ \exists\, \mathit{ma} \in \mathit{msgs} : \mathit{ma}.\mathit{type} = \text{“2a”} \land \mathit{ma}.\mathit{bal} = m.\mathit{bal} \land \mathit{ma}.\mathit{decrees} = m.\mathit{decrees}$
      $\land\ m.\mathit{bal} \leq \mathit{acceptorMaxBal}[m.\mathit{from}])'$
```

Induction for properties over sets, and ways of accessing elements of tuples. After developing the proof using the above strategy, we were still faced with certain assertions which were difficult to prove. One of the main difficulties lay in proving properties about tuples and sets of tuples for each of a set of processes in Multi-Paxos, as opposed to scalars for each of a set of processes in Basic Paxos. It may appear that, in many places, this requires simply adding an extra parameter for the slot, but the proof became significantly more difficult: even in places where an explicit inductive proof is not needed, auxiliary facts had to be added to help TLAPS succeed or proceed faster.

For example, adding slots to the proof of THEOREM *Consistent* for Basic Paxos caused the prover to take about 90 seconds to check it. To aid the proof, we added $\exists\, a \in \mathcal{A} : \ \mathit{VotedForIn}(a, v_1, b_1, s) \land \mathit{VotedForIn}(a, v_2, b_1, s)$ as an intermediary fact derivable from $\mathit{ChosenIn}(v_1, b_1, s) \land \mathit{ChosenIn}(v_2, b_2, s) \land b_1 = b_2$. Following this, the prover asserted the conclusion $v_1 = v_2$ in a few milliseconds.

Tuples have only a fixed number of components and therefore do not require separate inductive proofs, but they often turn out to be tricky and require special care in choosing the ways to access and test their elements, to reduce TLAPS's proof-checking time. For example, consider the definition of *VotedForIn* in Equation (6). Originally a test $[\mathit{slot} \mapsto s, \mathit{val} \mapsto v] \in m.\mathit{decrees}$ was written, because it was natural, but it had to be changed to $\exists\, d \in m.\mathit{decrees} : \ d.\mathit{slot} = s \land d.\mathit{val} = v$, because the prover found the latter more helpful. With the original version, the proof did not carry through after 1 or 2 minutes. After the change, the proof proceeded quickly. One minute of waiting for such simple, small tests felt very long, making it uncertain whether the proof would carry through.

5 Multi-Paxos with Preemption

Preemption is described informally in Lamport's description of Basic Paxos in Figure 1, in the paragraph about abandoning a proposal. Preemption has an acceptor reply to a proposer, in both Phases 1b and 2b, if the proposer's ballot is stale i.e., the acceptor has seen a higher ballot than the one just received from the proposer. This reply is a hint to the proposer to increase its ballot number.

To specify preemption, each of Phases 1b and 2b adds a new case for when the acceptor receives a lower ballot than some ballot it has seen before. We also define predicate *Preempt* that specifies how proposers update *proBallot* upon receiving a preemption message. Figure 6 shows Phase 1b with and without the modifications to add preemption. Modifications to Phase 2b are similar and are omitted for brevity.

Preemption adds a new phase in the variable *Next*, modifies definitions of existing phases, and adds a new type of message. This meant increasing the width of the proof tree for the new phase. This new branch of the proof was proven by asserting continuity lemmas already established earlier. The whole task of adding the new specification and proof took less than an hour.

6 Results of TLAPS-checked proof

Figure 7 summarizes the results from our specification and proof.

> The specification size grew by only 18 lines (16%), from 115 lines for Basic Paxos to 133 lines for Multi-Paxos; another 23 lines are added for Preemption.

```
$NewBallot(bb \in \mathcal{B}) \triangleq$ CHOOSE $b \in \mathcal{B}$ :
        $\land b > bb$
        $\land \nexists\, m \in msgs \; : \; m.type =$ "1a" $\land\, m.bal = b$

$Preempt(p \in \mathcal{P}) \triangleq \exists\, m \in msgs$ :
        $\land m.type =$ "preempt"
        $\land m.to = p$
        $\land m.bal > proBallot[p]$
        $\land proBallot' = [proBallot$ EXCEPT $![p] = NewBallot(m.bal)]$
        $\land$UNCHANGED $\langle msgs, accVoted, accMaxBal \rangle$
```

Phase 1b without Preemption	Phase 1b with Preemption
$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in msgs$: $\land m.type =$ "1a" $\land m.bal > accMaxBal[a]$ $\land Send([type \mapsto$ "1b", $bal \mapsto m.bal,$ $voted \mapsto accVoted[a],$ $from \mapsto a])$ $\land accMaxBal' =$ $[accMaxBal$ EXCEPT $![a] = m.bal]$ $\land$UNCHANGED $\langle accVoted, proBallot \rangle$	$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in msgs$: $\land m.type =$ "1a" $\land$IF $m.bal > accMaxBal[a]$ THEN $\land Send([type \mapsto$ "1b", $bal \mapsto m.bal,$ $voted \mapsto accVoted[a],$ $from \mapsto a])$ $\land accMaxBal' =$ $[accMaxBal$ EXCEPT $![a] = m.bal]$ $\land$UNCHANGED $\langle accVoted, proBallot \rangle$ ELSE $\land Send([type \mapsto$ "preempt", $to \mapsto m.from,$ $bal \mapsto acceptorMaxBal[a]])$ $\land$UNCHANGED $\langle accVoted, accMaxBal,$ $proBallot \rangle$

Fig. 6. Extension of Multi-Paxos to Multi-Paxos with Preemption

The proof size increased significantly by 763 lines (180%), from 423 for Basic Paxos to 1106 for Multi-Paxos, due to the complex interaction between slots and ballots; only 30 more lines are added for Preemption, thanks to the reuse of all lemmas, especially continuity lemmas.

The maximum level of proof tree nodes increased from 7 to 11 going from Basic Paxos to Multi-Paxos but remained 11 after adding Preemption; this contrast is even stronger for the maximum degree of proof tree nodes, consistent with challenge of going to Multi-Paxos.

The increase in number of lemmas is due to the change from *Maximum* in Basic Paxos to *MaxVotedBallotInSlot* in Multi-Paxos, defined in Equation (7). Five lemmas were needed for this predicate alone to aid the prover, as we moved from scalars to a set of tuples for each acceptor.

No proof by induction on set increment is used for Basic Paxos. Four such proofs are used for Multi-Paxos and for Multi-Paxos with Preemption.

Proof by contradiction is used once in the proof of Basic Paxos, and we extended it with slots in the proof of Multi-Paxos and Multi-Paxos with Premption.

The number of proof obligations to the prover increased most significantly, by 679 (284%), from 239 for Basic Paxos to 918 for Multi-Paxos. Only another 41 proof obligations were added for Multi-Paxos with Preemption.

The proof-checking time increased significantly, by 104 seconds, from 24 for Basic Paxos to 128 for Multi-Paxos, despite our continuous efforts to help the prover reduce it, because of the greatly increased size and complexity of the inductions used, leading to significantly more obligations to the prover. Going to Multi-Paxos with Preemption, however, the proof-checking time decreased by about 25%. This was initially surprising, but our understanding of Paxos and experience with proofs help support it: (1) adding the preemption cases to the original Phases 1b and 2b helps make the obligations in these cases more specialized and the remaining steps for proving consistency (which carry on longer in these cases before) easier; (2) adding the preemption action with Phases 1a and 2a increases the number of proof obligations, but the new obligations are easy, because they simply let the proposer start over (and thus there are no remaining steps in these cases). We are investigating further to confirm these.

Metric	Basic Paxos	Multi-Paxos	Multi-Paxos w/ Preemption
Size of specification (lines)	115	133	158
Size of proof (lines)	423	1106	1136
Max level of proof tree nodes	7	11	11
Max degree of proof tree nodes	3	17	17
# lemmas	4	11	12
# continuity lemmas	1	5	6
# uses of continuity lemmas	8	27	29
# proofs by induction on set increment	0	4	4
# proofs by contradiction	1	1	1
# obligations in TLAPS	239	918	959
Time to check by TLAPS (seconds)	24	128	94

Fig. 7. Summary of results. An obligation is a condition that TLAPS checks. The time to check is on an Intel i7-4720HQ 2.6 GHz CPU with 16 GB of memory, running Windows 10 and TLAPS version 1.5.2.

7 Related work and conclusion

We discuss closest related results on verification of Paxos, categorized by the verification technique.

Model checking. Model checking automatically explores the state space of systems [5]. Lamport wrote TLA+ specifications for Basic Paxos and its variants, e.g., Fast Paxos [20], and checked them using the TLA+ model checker TLC [25], but he has not done this for Multi-Paxos or its variants; a number of MS students at our university have also done this in course projects, including for Multi-Paxos. Delano et al. [7] modeled Basic Paxos in Promela and checked it using the Spin model checker [32]. To reduce the state space, they use counting guards to track majority, reset local variables after state operations, and use sorted *send* instead of FIFO *send* (with random *receive*, to model non-FIFO channels). They checked Basic Paxos for pairs of numbers of proposers and acceptors up to (2,8), (3,5), (4,4), (5,3), and (8,2). Yabandeh et al. [36] checked a C++ implementation of Basic Paxos using CrystalBall, a tool built on Mace [16], which includes a model checker. Yang et al. [37] used their model checker MoDist to check a Multi-Paxos-based service system developed by a Microsoft product team [24]. With dynamic partial-order reduction [9], they found 13 bugs including 2 bugs in the Paxos implementation, with as few as 3 replicas and a few slots. In all cases, existing work in model checking either does not check Multi-Paxos or can check it for only a very small number of slots and processes.

Deductive verification. Kellomaki [15] formally specified and verified Basic Paxos using PVS [33]. Jaskeliof and Merz [14] specified and verified a variant of Basic Paxos, called Disk Paxos [10], using Isabelle/HOL [34]. Charron-Bost and Schiper [4] expressed Basic Paxos in the Heard-Of model, and Charron-Bost and Merz [3] verified it formally using Isabelle/HOL. Drăgoi et al. [8] specified and verified a version of Basic Paxos in PSync, which is based on the Heard-Of model, so the specification and proof are similar to [4,3]. Lamport, Merz, and Doligez [23] give a formal specification of Basic Paxos in TLA+ and a TLAPS-checked proof of its correctness. Lamport [21] wrote a TLA+ specification of Byzantine Paxos, a variant of Basic Paxos that tolerates arbitrary failures, and a TLAPS-checked proof that it refines Basic Paxos. With IronFleet, Hawblitzel et al. [11] verified a state machine replication system that uses Multi-Paxos at its core. Their specification mimics TLA+ models but is written in Dafny [26], which has no direct concurrency support but has more automated proof support than TLAPS. This work is superior to its peers by proving not only safety but also liveness properties. Schiper et al. [31] used EventML [29] to specify Multi-Paxos and used NuPRL [6] to verify safety. Using the Verdi framework, Wilcox et al. [35] expressed Raft [28], an algorithm similar to Multi-Paxos, in OCaml and verified it using Coq [12]. All these works either do not handle Multi-Paxos or handle it using more restricted or less direct language models than TLA+, some mixed in large systems, making the essence of the algorithm's proof harder to find and understand.

In contrast, our work is the first to specify the exact phases of Multi-Paxos in a most direct and general language model, TLA+, with a complete correctness proof automatically checked using TLAPS. Building on Lamport, Merz, and Doligez's specification and proof for Basic Paxos [23], we aim to facilitate the understanding of Multi-Paxos and its proof by minimizing the difference from those for basic Paxos. We also show this as a general way for specifying and proving variants of Multi-Paxos, by doing so for Multi-Paxos extended with preemption. We also discuss the significantly more complex but necessary subproofs by induction. Future work may automate inductive proofs and support the verification of variants that improve and extend Multi-Paxos, by extending specifications of variants of Paxos, e.g., Fast Paxos [20] and Byzantine Paxos [21], to Multi-Paxos and verifying these variants of Multi-Paxos as well as Raft [28].

References

1. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. pp. 335–350. USENIX Association (2006)
2. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live—An engineering perspective. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing. pp. 398–407 (2007)
3. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the Heard-Of model. International Journal of Software and Informatics 3(2-3), 273–303 (2009)
4. Charron-Bost, B., Schiper, A.: The Heard-Of model: Computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
5. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
6. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall (1986)
7. Delzanno, G., Tatarek, M., Traverso, R.: Model checking Paxos in Spin. In: Proceedings of the 5th International Symposium on Games, Automata, Logics and Formal Verification. pp. 131–146 (2014)
8. Drăgoi, C., Henzinger, T.A., Zufferey, D.: Psync: A partially synchronous language for fault-tolerant distributed algorithms. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 400–415 (2016)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 110–121 (2005)
10. Gafni, E., Lamport, L.: Disk Paxos. Distributed Computing 16(1), 1–20 (2003)
11. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: Proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 1–17 (2015)
12. INRIA: The Coq Proof Assistant. `http://coq.inria.fr/` (Last released January 2016)
13. Isard, M.: Autopilot: Automatic data center management. ACM SIGOPS Operating Systems Review 41(2), 60–67 (2007)
14. Jaskelioff, M., Merz, S.: Proving the Correctness of Disk Paxos. Archive of Formal Proofs (June 2005), `http://isa-afp.org/entries/DiskPaxos.shtml`, Formal proof development
15. Kellomäki, P.: An annotated specification of the consensus protocol of Paxos using superposition in PVS. Report 36, Institute of Software Systems, Tampere University of Technology (2004)
16. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 179–188 (2007)

17. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
18. Lamport, L.: Paxos made simple. SIGACT News (Distributed Computing Column) 32(4), 51–58 (2001)
19. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
20. Lamport, L.: Fast Paxos. Distributed Computing 19(2), 79–103 (2006), http://research.microsoft.com/pubs/64624/tr-2005-112.pdf
21. Lamport, L.: Byzantizing Paxos by refinement. In: Proceedings of the 25th International Symposium on Distributed Computing. pp. 211–224. Springer (2011)
22. Lamport, L.: My writings. http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport (Accessed January 24, 2016), Lamport's history of paper [17]
23. Lamport, L., Merz, S., Doligez, D.: A TLA spefication of the Paxos Consensus algorithm described in Paxos Made Simple and a TLAPS-checked proof of its correctness. file /tlapm/examples/paxos/Paxos.tla in TLAPS distribution http://tla.msr-inria.inria.fr/tlaps/dist/current/tlaps-1.4.3.tar.gz (November 2012 Last modified November 28, 2014)
24. Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M.F., Zhang, Z.: D3S: Debugging deployed distributed systems. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. pp. 423–437. USENIX Association (2008)
25. Microsoft Research: The TLA Toolbox. http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox (Last modified January 4, 2016)
26. Microsoft Research: Dafny: A language and program verifier for functional correctness. http://research.microsoft.com/en-us/projects/dafny/ (Last released October 12, 2015)
27. Microsoft Research-Inria Joint Center: TLA+ Proof System (TLAPS). http://tla.msr-inria.inria.fr/tlaps/ (Last released June 2015)
28. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). pp. 305–319. USENIX Association (2014), http://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
29. PRL Project: EventML. http://www.nuprl.org/software/#WhatisEventML (Last released September 21, 2012)
30. van Renesse, R., Altinbuken, D.: Paxos made moderately complex. ACM Computing Surveys 47(3), 42:1–42:36 (Feb 2015)
31. Schiper, N., Rahli, V., van Renesse, R., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 395–406. IEEE CS Press (2014)
32. Spin Community: Verifying Multi-threaded Software with Spin. http://spinroot.com/spin/whatispin.html (Last released January 1, 2016)
33. SRI: PVS Specification and Verification System. http://pvs.csl.sri.com/ (Last released February 11, 2013)
34. University of Cambridge: Isabelle (a generic proof assistant). http://isabelle.in.tum.de/ (Last released May 25, 2015)
35. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 357–368 (2015)

36. Yabandeh, M., Knezevic, N., Kostic, D., Kuncak, V.: CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. pp. 229–244. USENIX Association (2009)
37. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MoDist: Transparent model checking of unmodified distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. pp. 213–228. USENIX Association (2009)

A Additional Predicates and Invariants

This appendix defines predicates and invariants not defined above due to space limits.

Predicates. Predicate $SafeAt(v \in \mathcal{V}, b \in \mathcal{B}, s \in \mathcal{S})$ means that no value other than perhaps v has been or will be chosen for any ballot lower than b for slot s:

$$\begin{aligned}
&WontVoteIn(a \in \mathcal{A}, b \in \mathcal{B}, s \in \mathcal{S}) == \\
&\quad \land \forall\, v \in \mathcal{V} : \neg\, VotedForIn(a, v, b, s) \\
&\quad \land\, accMaxBal[a] > b \\
&SafeAt(v \in \mathcal{V}, b \in \mathcal{B}, s \in \mathcal{S}) == \\
&\quad \forall\, c \in [0, b) : \exists\, Q \in \mathcal{Q} : \\
&\qquad \forall\, a \in Q : VotedForIn(a, v, c, s) \lor WontVoteIn(a, c, s)
\end{aligned} \tag{12}$$

Invariants about acceptors. For each acceptor, the first conjunct establishes the initial condition. The second conjunct says that each record in *accVoted* has been voted for by the acceptor. The third conjunct states the other direction: if voted for, the acceptor must have a record for that slot in its *accVoted*. The last conjunct says that an acceptor does not vote for a value in any ballot higher than the highest it has seen per slot.

$$\begin{aligned}
&AccInv == \\
&\quad \forall\, a \in \mathcal{A} : \\
&\qquad \land\, (accMaxBal[a] = -1) \Rightarrow (accVoted[a] = \{\}) \\
&\qquad \land\, \forall\, t \in accVoted[a] : VotedForIn(a, t.val, t.bal, t.slot) \\
&\qquad \land\, \forall\, b \in \mathcal{B}, s \in \mathcal{S}, v \in \mathcal{V} : \\
&\qquad\quad VotedForIn(a, v, b, s) \Rightarrow \exists\, t \in accVoted[a] : t.slot = s \land t.bal >= b \\
&\qquad \land\, \forall\, b2 \in \mathcal{B}, s \in \mathcal{S}, v \in \mathcal{V} : \\
&\qquad\quad b2 > MaxVotedBallotInSlot(accVoted[a], s) \Rightarrow \neg\, VotedForIn(a, v, b2, s)
\end{aligned} \tag{13}$$

Invariants about messages. The following invariant is for 2a messages. The first conjunct establishes safety on all decrees contained. The second conjunct says that for each slot, there is at most one decree in the message. The third conjunct states that ballot numbers uniquely identify 2a messages, i.e., at most one 2a message is sent for each ballot.

$$\begin{array}{l} MsgInv2a == \\ \quad \forall\, m \in msgs : (m.type = \text{”}2a\text{”}) \Rightarrow \\ \qquad \land\, \forall\, d \in m.decrees : SafeAt(d.val, m.bal, d.slot) \\ \qquad \land\, \forall\, d1, d2 \in m.decrees : d1.slot = d2.slot \Rightarrow d1 = d2 \\ \qquad \land\, \forall\, m2 \in msgs : (m2.type = \text{”}2a\text{”}) \land (m2.bal = m.bal) \Rightarrow m2 = m \end{array} \tag{14}$$

The following invariant is for 2b messages. The first conjunct states that a 2a message is needed for a 2b reply. The second conjunct asserts that the *bal* of this 2a message is at most the *accMaxBal* for this acceptor.

$$\begin{array}{l} MsgInv2b == \\ \quad \forall\, m \in msgs : (m.type = \text{”}2b\text{”}) \Rightarrow \\ \qquad \land\, \exists\, m2 \in msgs : m2.type = \text{”}2a\text{”} \land m2.bal = m.bal \land m2.decrees = m.decrees \\ \qquad \land\, m.bal =< accMaxBal[m.from] \end{array} \tag{15}$$

B TLA+ specification and TLAPS-checked proof of Multi-Paxos with Preemption