ABSTRACT

This paper describes the application of a high-level language and method in developing simpler specifications of more complex variants of the Paxos algorithm for distributed consensus. The specifications are for Multi-Paxos with preemption and reconfiguration and optimized with state reduction and failure detection. The language is DistAlgo. The key is to express complex control flows and synchronization conditions precisely at a high level, using nondeterministic waits and message-history queries. We obtain complete executable specifications that are almost completely declarative—updating only a number for the protocol round besides the sets of messages sent and received.

We show the following results: (1) English and pseudocode descriptions of distributed algorithms can be captured completely and precisely at a high level, without adding, removing, or reformulating algorithm details to fit lower-level, more abstract, or less direct languages. (2) We created higher-level control flows and synchronization conditions than all previous specifications, and obtained specifications that are much smaller and more precise, even when matching or smaller than abstract specifications that omit many algorithm details. (3) The simpler specifications led us to easily discover useless replies, unnecessary delays, and liveness violations (if messages can be lost) in a previous published specification, by just following the simplified algorithm flows. (4) The resulting specifications can be executed directly, and we can express optimizations cleanly, yielding drastic performance improvement over naive execution and facilitating a general method for merging processes. (5) We systematically translated the resulting specifications into TLA+ and developed machine-checked safety proofs, which also allowed us to detect and fix a subtle safety violation in an earlier specification. Additionally, we show the basic concepts in Paxos that are fundamental in many distributed algorithms and show that they are captured concisely in our specifications.

1 INTRODUCTION

Distributed algorithms are increasingly important as distributed applications are increasingly needed. These algorithms describe how distributed processes compute and communicate with each other by passing messages. However, because processes can fail and messages can be lost, delayed, reordered, and duplicated, distributed algorithms are often difficult to understand, even if they might appear simple. The most important and well-known of such algorithms is Paxos [23, 24, 44] for distributed consensus—a set of distributed processes trying to agree on a single value or a continuous sequence of values, called single-value consensus or multi-value consensus, respectively.

Distributed consensus and Paxos. Distributed consensus is essential in any important service that must maintain a state, including many services provided by companies like Google and Amazon. This is because such services must use replication to tolerate failures caused by machine crashes, network outage, etc. Replicated processes must agree on the state of the service or the sequence of operations that have been performed, e.g., a customer order has been placed and paid but not yet shipped, so that when some processes become unavailable, the remaining processes can continue to function correctly. Distributed consensus is exactly this agreement problem.

Many algorithms and variations have been proposed for distributed consensus, starting from Virtual Synchrony (VS) by Birman and Joseph [4] and Viewstamped Replication (VR) by Oki and Liskov [39]. These algorithms share similar ideas, but Paxos became the most well-known name and the focus of studies starting with its elaborate description by Lamport [23, 26]. The basic ideas in these algorithms are fundamental for not only consensus, but all distributed algorithms that must deal with replication, asynchronous communication, and failures. These ideas include successive rounds, a.k.a. views in VR [39] and ballots in Paxos [23], leader election; voting by majority or quorum; preemption; dynamic reconfiguration; state reduction; and failure detection by periodic probe or heartbeat. Van Renesse et al. give extended discussions of these ideas [44] and comparisons of major variants of Paxos [45].

Paxos is well-known to be hard to understand [24]. Since Paxos was introduced by Lamport [23, 26], there has been a continuous series of studies of it. This includes not only optimizations and extensions, especially for use in practical systems, e.g., Google’s distributed lock service Chubby [7, 10], but also more variations and expositions, especially with effort for better understanding, e.g., [11, 15, 24, 28, 40, 44, 45], and for formal verification, e.g., [8, 9, 17, 42, 43, 46, 47]. Major developments in this series have led to a better comprehensive understanding of Paxos, as presented in Paxos Made Moderately Complex by van Renesse and Altinbukken [44], starting from its simpler core, as presented in Paxos Made Simple by Lamport [24]. Can these algorithms be made completely precise, readily executable in real distributed environments, and at the same time easier to understand?

This paper. This paper describes the application of a high-level language and method in developing simpler specifications of both
We also show the basic concepts in Paxos that are fundamental and more complex variants of the Paxos algorithm for distributed consensus. The specifications are for Paxos for single-value consensus, as described by Lamport in English [24], which we call Basic Paxos, and Paxos for multi-value consensus with preemption and reconﬁguration, as described by van Renesse and Altinbıken in pseudocode [44], which we call vRA Multi-Paxos, as well as vRA Multi-Paxos optimized with state reduction and failure detection [44]. The key is to express complex control ﬂows and synchronization conditions precisely at a high level, using non-deterministic waits and message-history queries. We obtain complete executable speciﬁcations that are almost completely declarative—updating only a number for the protocol round besides the sets of messages sent and received. Our contributions include the following:

- We show that English and pseudocode descriptions of algorithms can be captured completely and precisely at a high level, without adding, removing, or reformulating algorithm details to ﬁt lower-level, more abstract, or less direct languages.
- We created higher-level control ﬂows and synchronization conditions than all previous speciﬁcations of these algorithms, and obtained speciﬁcations that are much simpler and smaller, even matching or smaller than abstract speciﬁcations that omit many algorithm details.
- We show that the simpler speciﬁcations led us to easily discover useless replies, unnecessary delays, and liveness violations (if messages can be lost) in the original vRA Multi-Paxos speciﬁcation and in our earlier speciﬁcations, by just following the simpliﬁed algorithm ﬂows.
- We demonstrate that the resulting speciﬁcations can be executed directly, and we can express optimizations cleanly, yielding drastic performance improvement over naive execution and facilitating a general method for merging processes.
- We systematically translated the resulting speciﬁcations into TLA+ [25] and developed machine-checked safety proofs, which also allowed us to detect and ﬁx a subtle safety violation in an earlier speciﬁcation.

We also show the basic concepts in Paxos that are fundamental in many distributed algorithms and show that they are captured concisely in our speciﬁcations.

Our speciﬁcations are written in DistAlgo [32], a language for distributed algorithms with a formal operational semantics [34] and a complete implementation in Python [31, 35]. Our complete executable speciﬁcations in DistAlgo and machine-checked proofs in TLAPS are available at darlab.cs.stonybrook.edu/paxos.

There have been numerous studies of speciﬁcations for understanding and veriﬁcation of distributed algorithms, especially of Paxos, as discussed in Section 7. With the exception of vRA Multi-Paxos, no previous papers present direct, complete, and precise speciﬁcation of Multi-Paxos. Previous formal speciﬁcations are only for Basic Paxos, e.g., [3, 19], are abstract by omitting many algorithm details necessary for real execution, e.g., [9, 42], or are too long or too complicated to include in papers, e.g., [17, 47]. Also, to the best of our knowledge, no previous efforts of speciﬁcation and formal veriﬁcation, for any Paxos variant, reported ﬁnding any correctness violations or improvements. Nor did previous efforts of implementation of vRA Multi-Paxos in several languages, including in Python [44].

2 BASIC PAXOS, LANGUAGE, AND HIGH-LEVEL SPECIFICATION

We describe the language, DistAlgo, and method of speciﬁcation using Basic Paxos as an example. Prior knowledge of Paxos can be helpful, but the description is self-contained.

Paxos considers distributed processes that may crash and may later recover, and messages that may be lost, delayed, reordered, and duplicated. It guarantees safety, i.e., agreement on the decided single value in Basic Paxos (or sequence of values in Multi-Paxos) by non-faulty processes, and validity of the decided value (or values) to be among allowed values. However, it does not guarantee liveness, i.e., non-faulty processes eventually decide, without stronger assumptions, due to the well-known impossibility result [13].

2.1 Basic Paxos in English

Figure 1 shows Lamport’s description of Basic Paxos in English [24]. It presents the algorithm for (1) the proposer and acceptor—the two phases, and (2) the learner—the obvious algorithm. From it, we can see that high-level speciﬁcation of distributed algorithms needs four main concepts:

1. Distributed processes that can send messages. In Figure 1, there are proposer, acceptor, and learner processes, prepare and accept messages from a proposer to an acceptor, response messages back from an acceptor, and messages for accepted proposals from an acceptor to a learner.

2. Control ﬂows for handling received messages. In Figure 1, messages can be received by acceptors, proposers, and learners asynchronously at any time, but processes must synchronize by testing and waiting for different conditions on the received messages. Capturing such complex control ﬂows is essential.

3. High-level queries for synchronization conditions. In Figure 1, the conditions checked in Phases 1b, 2a, 2b, and the learner before taking actions involve sets of many messages sent and received. Capturing such conditions at a high level is the most important key to making control ﬂows much clearer and easier to understand.

4. Conﬁguration for setting up and running. This is often implicit in descriptions of distributed algorithms. In Figure 1, each process needs to be set up and get a hold of other processes with which it needs to communicate. In general, there may also be special conﬁguration requirements, such as use of speciﬁc logical clocks.

Figure 2 shows a complete high-level executable speciﬁcation of Basic Paxos in DistAlgo, including setting up and running. It will be explained in examples for the language in Section 2.2 and discussed in more detail in Section 2.3. The overall structure has two main aspects:

- Figure 1 corresponds to the body of run in Proposer (lines 5-13), the two receive deﬁnitions in Acceptor (lines 17-24), and the wait condition in Learner (lines 28-30), including selecting “a proposal number” in Phase 1a to be self (line 5),
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 \( n \).

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.

**Figure 1:** Lamport’s description of Basic Paxos in English [24].

**Figure 2:** A high-level specification of Basic Paxos in DistAlgo, including setting up and running 3 each of Proposer, Acceptor, and Learner processes and outputting the result.
as is commonly used, and taking "any value" in Phase 2a to be any integer in 1..100 as the set of allowed values (line 11), for simplicity.

- The rest puts all together, plus setting up processes, starting them, and outputting the result of the execution.

Note that Figure 1, and thus Figure 2, specifies only one round of the two phases for each process. Section 2.3 discusses successive rounds that help increase liveness.

2.2 Language and high-level specification

We use DistAlgo, a language that supports the four main concepts in Section 2.1 by building on an object-oriented programming language, with a formal operational semantics [34].

Besides the language constructs explained below, commonly used notations in high-level languages are used for no operation (pass), assignments (:= e), etc. Indentation is used for scoping, "." for separation, and "*" for comments.

Distributed processes that can send messages. A type \( P \) of distributed processes is defined by process \( P \): `body`, e.g., lines 1-13 in Figure 2. The body may contain

- a setup definition for taking in and setting up the values used by a type \( P \) process, e.g., lines 2-3,
- a run definition for running the main control flow of the process, e.g., lines 4-13, and
- receive definitions for handling received messages, e.g., lines 17-21.

A process can refer to itself as self. Expression `self. attr` (or `attr` when there is no ambiguity) refers to the value of `attr` in the process. `ps := n new P(args)` creates \( n \) new processes of type \( P \), optionally passing in the values of `args` to `setup`, and assigns the new processes to `ps`, e.g., lines 33 and 34. `ps.setup(args)` sets up processes `ps` using values of `args`, e.g., line 36, and `ps.run()` starts `run()` of processes `ps`, e.g., line 37.

Processes can send messages: `send m to ps` sends message `m` to processes `ps`, e.g., line 6.

Control flow for handling received messages. Received messages can be handled both asynchronously, using `receive` definitions, and synchronously, using `await` statements.

- A definition, `receive m from p` handles, at yield points, unhandled messages that match `m` from `p`, e.g., lines 17-21. There is a yield point before each `await` statement, e.g., line 7, for handling messages while waiting. The `from` clause is optional, e.g., line 22.

  - A statement, `await cond1; stmt1 or ... or condk; stmtk` timeout: `stmt`, waits for one of `cond1`, ..., `condk` to be true or a timeout after period \( z \), and then nondeterministically selects one of `stmt1`, ..., `stmtk`, `stmt` whose conditions are true to execute, e.g., lines 7-13. Each branch is optional. So is the statement in `await` with a single branch.

High-level queries for synchronization conditions. High-level queries can be used over message histories, and patterns can be used for matching messages.

- Histories of messages sent and received by a process are kept in variables `sent` and `received`, respectively. `sent` is updated at each `send` statement, by adding each message `sent`. `received` is updated at yield points, by adding un-handled messages before executing all matching `receive` definitions.

Expression `sent m to p` is equivalent to `m` to `p` in `sent`. It returns true iff a message that matches `m` to `p` is in `sent`. The `to` clause is optional. `received m from p` is similar.

- A pattern can be used to match a message, in `sent` and received, and by a `receive` definition. A constant value, such as `response`, or a previously bound variable, indicated with `i`, in the pattern must match the corresponding components of the message. An underscore matches anything. Previously unbound variables in the pattern are bound to the corresponding components in the matched message. For example, `received (‘response’, =i, …)` from `on` line 7 matches every triple in `received` whose first two components are `response` and the value of `n`, and binds `a` to the sender.

A query can be a comprehension, aggregation, or quantification over sets or sequences.

- A comprehension, `(e: v1 in s1, ..., vn in sn, cond)`, where `v` can be a pattern, returns the set of values of `e` for all combinations of values of variables that satisfy all `vi` in `si` clauses and condition `cond`, e.g., the comprehension on line 7.

- An aggregation, `agg s`, where `agg` is an aggregation operator such as `count` or `max`, returns the value of applying `agg` to the set value of `s`, e.g., the `count query` on line 7.

- An existential quantification, `some v1 in s1, ..., vn in sn has cond`, returns true iff for some combinations of values of variables that satisfy all `vi` in `si` clauses, `cond` holds, e.g., the `some query` on line 23. When the query returns true, all variables in the query are bound to a combination of satisfying values, called a witness, e.g., `s` and `v` on lines 28-30.

- A universal quantification, `each vi in s1, ..., vn in sn has cond`, returns true iff for all combinations of values of variables that satisfy all `vi` in `si` clauses, `cond` holds, e.g., the `each query` on line 18.

Other operations on sets can also be used, in particular:

- any `s` returns any element of set `s` if `s` is not empty, and a special value `undefined` otherwise.

- `n1..n2` returns the set of integers ranging from `n1` to `n2` for `n1 ≤ n2`.

- `s1 + s2` returns the union of sets `s1` and `s2`.

- `s1 or s2` returns `s1` if `s1` is not empty, and `s2` otherwise.

Configuration for setting up and running. Configuration for requirements such as logical clocks can be specified in a `main` definition, e.g., lines 32-37. Basic Paxos does not have special configuration requirements, besides setting up and running the processes by calling `new`, `setup`, and `start` as already described. In general, `new` can have an additional clause `at node` specifying remote nodes where the created processes will run; the default is the local node.

High-level specification of distributed algorithms via declarative queries. The core of DistAlgo supports, besides distributed
processes that can send and receive messages, prominently high-level constructs for expressing complex control flows and synchronization conditions, using nondeterministic waits with message-history queries. These constructs are not supported in other languages for concurrent and distributed processes, including Erlang [12, 29] and languages like CSP [18] and CCS [38].

Examine the Basic Paxos algorithm specification in Figure 2, in processes for Proposer, Acceptor, and Learner. One can see that all variables in assignment statements are either assigned only once (majority and n) or assigned temporarily to be consumed immediately (v, responded, and max_prop). In other words, these variables are like those bound in let expressions in functional languages. Therefore, there are no intrinsic state updates besides sending and receiving messages that update the built-in variables sent and received.

Sending and receiving messages are essential in distributed algorithms. However, the rest of the algorithm can be specified declaratively, by using high-level queries over sent and received, and using them in await conditions for synchronization and in if conditions for safeguarding. This is the key to making specifications of distributed algorithms high-level and declarative, and therefore easier to understand.

2.3 Understanding Basic Paxos and fundamentals of distributed algorithms

For Basic Paxos, we now see how Phases 1 and 2 in Figure 2 precisely follow Lamport’s description in Figure 1.

Phase 1a (lines 5-6). This straightforwardly follows the description in Figure 1. A proposal number can be any value that allows the comparison operation >.

Phase 1b (lines 17-21). When an acceptor receives a prepare message with a proposal number larger than all numbers in its previous responses, it responds with this proposal number and with any sent accepted (n,v) pair where n is maximum among all such pairs it has sent in accepted messages; note that, if it has not sent any accepted messages, max_prop is undefined, instead of some (n,v), in the response message.

Phase 2a (lines 7-13). When a proposer receives responses to its proposal number n from a majority of acceptors, it takes v in the (n2,v) that has the maximum n2 in all responses to n, or any value in allowed values 1..100 if the responses contain no (n2,v) pairs but only undefined; note that, in the latter case, the set on lines 9-10 is empty because undefined does not match (n2,v). The proposer then sends accept for a proposal with number n and value v to acceptors that responded to n.

Phase 2b (lines 22-24). This directly follows the description in Figure 1.

In particular, the specification in Figure 2 makes the "promise" in Phase 1b of Figure 1 precise—the "promise" refers to the responded proposal number that will be used later to not accept any proposal with a smaller proposal number in Phase 2b.

Indeed, this is the hardest part for understanding Paxos, because understanding the respond message sent in Phase 1b requires understanding the accepted message sent in Phase 2b, a later phase, but understanding the later phase depends on understanding the earlier phase. The key idea is that the Phase 2b to be understood is for a smaller proposal number, i.e., the accepted messages used in Phase 1b are those sent with smaller proposal numbers than the n in received (‘prepare’, n). For the smallest n, no accepted messages have been sent, and Phase 1b simply responds with max_prop being undefined.

Fundamental concepts in distributed algorithms. Basic Paxos contains several concepts that are fundamental in distributed algorithms and are commonly used:

- **Leader election.** This is as done in Phase 1 but on only lines 5-6, 17-18, and 21 (ignoring lines 19-20). It is for electing at most one proposer (with its proposal number n) at a time as the leader. It ends with the await condition on lines 7-8 taking a majority. In Basic Paxos, only after receiving responses from a majority does a proposer carry out Phase 2 and propose to accept a value v.

- **Majority or quorum voting.** This is as done on lines 7-8 and 29-30 that test with a majority. It ensures that at most one value is in the voting result. In Basic Paxos, the two votings are for electing a leader n on line 7 and choosing an accepted proposal n, v on line 29.

- **Successive rounds.** This is as partially done on lines 5, 18, and 23, with larger proposal numbers taking over smaller ones. Rounds are used to make progress with increasingly larger numbers. To do this fully to increase liveness, each proposer may iterate (i.e., repeat the body of run if its await may fail for any reason), with a larger number n in each iteration. We will see this in Multi-Paxos in Section 3, by using a pair for the proposal number, called ballot number there; it is the only intrinsic state variable, besides sent and received, in Multi-Paxos.

- **Timeout and failure detection.** To fundamentally increase liveness, each proposer may add a timeout to its await, or an alternative branch in await to receive messages indicating a preemption of the condition originally waited for. Time-out is the most important mechanism to provide liveness in practice. We will see preemption in Multi-Paxos in Section 3, and see timeout and failure detection in Section 5.

- **Selection with maximum or minimum.** This is as done for computing max_prop and v on lines 19-20 and 9-11, by using maximum to select over collections. In Basic Paxos, this is for passing on the agreed value (once voted by a majority in a round) to successive later rounds.

These are all made simple and precise by our high-level control flows and synchronization conditions, especially with declarative queries over message history variables sent and received.

To summarize, uses of high-level control flows and declarative queries allow our Basic Paxos specification to be at the same high level as Lamport’s English description, while making everything precise. With also precise constructs for setting up and running, the complete specification is both directly executable in real distributed environments, by automatic compilation to a programming language, and ready to be verified, by systematic translation to a verifier language.
3 MULTI-PAXOS WITH PREEMPTION AND RECONFIGURATION, AND HIGH-LEVEL SPECIFICATION

Multi-Paxos extends Basic Paxos to reach consensus on a continuing sequence of values, instead of a single value. It is significantly more sophisticated than running Basic Paxos for each slot in the sequence, because proposals must be made continuously for each next slot, with the proposal number, also called ballot number or simply ballot, incremented repeatedly in new rounds if needed, and the ballot is shared for all the slots for obvious efficiency reasons.

Preemption allows a proposer, also called leader, to be preempted by another leader that has a larger ballot, i.e., if a leader receives a message with a larger ballot than its own ballot, it abandons its own and uses a larger ballot later.

Reconfiguration allows switching to a set of new leaders during execution of the algorithm. The slot in which the change of leaders is to happen must be agreed on by the old leaders. This is done by taking the change as one of the values, also called commands, to be agreed on. Note that the new leaders can be set up with a new set of acceptors.

We present a complete specification of Multi-Paxos with preemption and reconfiguration, developed based on vRA Multi-Paxos. The specification improves over the original pseudocode in several ways and also led us to easily discover liveness violations when messages can be lost.

3.1 vRA Multi-Paxos pseudocode

vRA Multi-Paxos gives complete pseudocode for Multi-Paxos with preemption and reconfiguration [44]; note it additionally includes replicated state machines for applications. The core ideas are the same as Basic Paxos. However, except for the name Acceptor, all other names used, for processes, data, and message types, are changed:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Process Types</th>
<th>Data Types</th>
<th>Message Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Paxos</td>
<td>Proposer, Learner</td>
<td>proposal number</td>
<td>prepare, respond, accept, accepted</td>
</tr>
<tr>
<td>vRA Multi-Paxos</td>
<td>Leader, Scout, Commander</td>
<td>ballot number</td>
<td>command 1a, 1b, 2a, 2b</td>
</tr>
</tbody>
</table>

or new:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Process Types</th>
<th>Data Types</th>
<th>Message Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRA Multi-Paxos</td>
<td>Replica</td>
<td>slot</td>
<td>request, response, propose, decision, preempt</td>
</tr>
</tbody>
</table>

- A Leader process spawns Scout and Commander processes to perform Phases 1 and 2, respectively, on its behalf; the spawned processes also determine preemption and inform the Leader process (preempt message). These three kinds of processes together have essentially the same role as Proposer and Learner, except that they also handle slots.
- A Replica process keeps the state of an application, e.g., a bank. It repeatedly receives a requested command (request) from a client, and sends a proposed slot for the command (propose) to Leader processes; it also receives a decided slot for a command (decision), applies the operation in the command to the state at the decided slot, and sends the result (response) to the client.
- A slot is just a component in a proposal or decision in Leader and Acceptor, but is tracked in Replica using variables slot_in and slot_out, for the next slot to send a proposal and the next slot to apply a decision, respectively. A window between slot_in and slot_out is used so that a decided reconfiguration at a slot takes effect at the slot at the end of the window, while other commands can still be proposed and decided for slots within the window.
- A command is a triple of client id, command id, and operation, and the operation for reconfiguration holds the set of new leaders.

The complete pseudocode (Figs. 1, 4, 6, 7, and the two formulas on pages 6, 9, 12, 13, and 14 in [44]) is precise and succinct, even though not directly executable. Appendix A shows the pseudocode for the Leader process, the center of the algorithm. However, there are two main challenges in understanding the overall algorithm:

1. Each of the 5 kinds of processes maintains additional process state variables. These variables are updated repeatedly or in multiple places in the process without explicit invariants or properties about the values in the variables.
2. Each of the 5 kinds of processes contains an infinite loop. The body of the loop is driven by receiving a message and performing the associated actions without expressing higher-level conditions over the messages sent or received.

For example, a Replica process maintains 3 sets: requests, proposals, and decisions. Set requests is (i) initialized to empty, (ii) added to after receiving a request message, (iii) deleted from under a condition about decisions in a while loop in the top-level infinite loop, and (iv) added to under two nested conditions inside a while loop after receiving a decision message.

3.2 Higher-level executable specification

To overcome the challenges in understanding the algorithm, we worked hard to find the hidden properties and developed higher-level control flows and synchronization conditions.

Figure 3 shows a complete higher-level executable specification of vRA Multi-Paxos in DistAlgo. Even though executable code is generally much longer and more complex than pseudocode, our specification is smaller than vRA Multi-Paxos pseudocode (51 vs. 142 lines, or 100 lines without "end" of 42 scopes). More importantly, it is much simpler. The new organization and main simplifications are as follows. Other improvements are described in Section 4, after understanding the overall algorithm better in Section 3.3.
Figure 3: A high-level specification of vRA Multi-Paxos in DistAlgo.

Scout and Commander are removed. Their roles for Phases 1 and 2 are merged into Leader. Their roles for determining preemption are merged into Acceptor.

Repeated and scattered updates are replaced by high-level queries. Leader collects majority from Phases 1b and 2b using two count queries (lines 27 and 35), not repeated updates of two waitfor sets in Scout and Commander; finds previously accepted and newly proposed proposals using only comprehensions and some queries (lines 28-29 and 32), not maintaining and updating a proposals set; and confirms preemption using some queries (lines 37 and 38), not maintaining an overall variable active and updating it to false here and true elsewhere.

Accepter uses each and some (lines 43 and 47) and a comprehesion (line 44), instead of using and maintaining a maximum ballot num and an accepted set by updates; and it determines preemption using a max query (line 50) in one place, instead of always sending 1b and 2b messages to scouts and commanders and letting them determine preemption.

Replica uses two clearly separated conditions, one for proposing commands based on requests, and one for applying commands based on decisions, instead of maintaining requests,
proposals, and decisions by additions and deletions in mixed control flows and loop structures.

High-level specification via declarative queries. Examine the entire algorithm specification in Figure 3.

- In process Replica for replicated state machines with reconfiguration, there are of course state variables updated for the application state (slot_in, slot_out, and state) and reconfiguration state (leader); they are orthogonal to the consensus algorithm run by Leader and Acceptor processes. All other variables in assignments (client, cmd_id, op, and result) are only temporary variables to be consumed immediately.

- In processes Leader and Acceptor for Multi-Paxos with preemption, there is only one state variable, ballot, and it is repeatedly updated. All other variables in assignments (ps, accepted, and maxb) are only temporary variables to be consumed immediately.

Updating state in replicated state machines for applications is of course essential, just as sending and receiving messages are for distributed algorithms. However, the rest of the algorithm is specified almost completely declarative, by using declarative queries over sent and received.

The only exception is the update to ballot, which identifies successive rounds, as discussed in Section 2.3. This minimum state for tracking progress is fundamental in distributed systems, just like using logical clocks [22].

3.3 Understanding Multi-Paxos with preemption and reconfiguration

Without low-level updates, we now see how Figure 3 precisely extends Basic Paxos with continuous slots, preemption, and Replica with reconfiguration.

A Leader process takes a set acceptors and a set replicas (line 22), initializes ballot to round 0 paired with self (line 24), and repeatedly (line 25) does two things with each incremented ballot (line 39).

1. Send 1a message for the current ballot to acceptors (line 26) and wait for a majority 1b replies for the ballot (line 27), as in Proposer in Phase 1 of Basic Paxos; but also wait for some preempt with a larger ballot (r2, leader2) than the ballot and then do nothing more for the current ballot (line 38). Note that between receiving a majority 1b messages and receiving a preempt is exactly when the original pseudocode maintains active as true for the current ballot.

2. After receiving a majority 1b replies, (1) find in 1b messages (line 28) previously accepted slot s and command c pairs that correspond to the largest ballot for each s (line 29) and send them with the current ballot to acceptors (line 30), ensuring that they continue to be accepted in the current ballot, and (2) repeatedly (line 31) wait to (i) receive newly proposed s and c for which no 2a message has been sent for s for the ballot (line 32) and send a 2a message for s and c to acceptors (line 33), or (ii) receive a majority 2b replies for some s and c (lines 34-35) and send s and c as a decision to replicas (line 36), or (iii) receive a preempt with a larger ballot than the ballot and break out of the repeats.

This is as in Proposer in Phase 2 of Basic Paxos except that a 1b message holds a set of triples instead of a single pair; a 2a message is sent for a triple (ballot, s, c) for each slot s instead of a single pair (n, v); a 2b message has an additional slot component too, and is received by the leader instead of a learner, and a decision is sent back to replicas instead of a “learned” being outputted; and receiving a preempt message is added.

An Acceptor process continuously waits (line 41) to receive 1a and 2a messages and does one of two things:

- Use each and some queries to check sent 1b messages (lines 43 and 47), and send back 1b and 2b messages (lines 45 and 48), exactly as in Acceptor in Phases 1 and 2 of Basic Paxos except with the additional s component in 2a and 2b messages, and computing a set accepted of proposals instead of a single max_prop proposal.

- Test if the ballot in a received 1a or 2a message is less than the maximum ballot ever received (line 50), and send back preempt with the maximum ballot (line 51). This is not only much simpler and more direct, it is also much more efficient than always sending 1b messages with the set accepted that is large and will be ignored anyway.

A Replica process takes a set leaders and a state (line 2), initializes slot_in and slot_out to 1 (line 4), and repeatedly (line 5) executes one of two branches and increments slot_in or slot_out (lines 14 and 20):

- If some received requested command c (line 7) is such that each sent proposed slot s for c (line 8) has s taken by a different command c2 in some received decision (line 9), propose slot_in for c to leaders (line 13) if slot_in is not already used in some received decision (line 12). For reconfiguration, check also that slot_in is within WINDOW slots ahead of slot_out (line 6), and if the decision at WINDOW slots back is a command for a reconfiguration operation (line 10), set new leaders to be those in that operation (line 11).

- If some received decided command c is for slot_out (line 15), get the client, command id, and operation in c (line 16), and if there was not already a decision for c in some earlier slot s and the operation is not reconfiguration (line 17), apply the operation to state to obtain a new state and a result (line 18), and send the result with the command id to the client (line 19).

The complete preemption functionality is expressed simply on lines 30-31 and 37-38, and reconfiguration is completely expressed on lines 6 and 10-11 with two increments on lines 14 and 20. These are easy to see, again due to our use of high-level queries for control flows and synchronization conditions.

4 ISSUES AND FIXES

This section describes useless replies, unnecessary delays, liveness violations and fixes discovered in developing the specification in Figure 3. All problems to be described were difficult to find in the original pseudocode due to complex control flows. Developing higher-level specifications, especially using nondeterministic await with message-history queries for synchronization conditions,
helped us understand the algorithm better and discover these problems easily, by just following the simplified algorithm flows.

The liveness violations can occur if messages can be lost. The liveness violation in Replica was confirmed by author van Renesse when we first discovered it. However, we recalled later that the paper has used a fair links assumption. It is a strong assumption, and assumes that messages are periodically retransmitted until an ack is received. Nonetheless, implementing such retransmission is not easy without slowing down the executions too much, because even TCP connections can break, and retransmission would need to include repairing broken TCP connections. The Ovid framework by the authors [2] does such repair and retransmission automatically. When we discovered the liveness violation in Leader, we learned from author van Renesse that making a framework like Ovid efficient is very difficult. We describe fixes without relying on message retransmission.

Useless replies in Acceptor are fixed. In Figure 3, if an acceptor receives a 2a message with a larger ballot b than the maximum ballot in all sent 1b messages, i.e., all received 1a messages, it replies with the same ballot b (line 48), which will be used when counting majority for ballot b, exactly as in Basic Paxos. In the original pseudocode, it replies with that maximum (line 15 in Figure 4 in [44]), smaller than b, causing the reply to be ignored by the Commander and Leader processes, rendering the reply useless.

Unnecessary delays in Replica are fixed. In Figure 3, the first await condition (lines 6-9) allows any received request, for which each sent proposal for a slot has that slot taken by a different command in a received decision, to be detected and re-proposed immediately. The original pseudocode delays the detection and re-proposal until slot_out equals the taken slot.

This delay could be relatively minor by itself, but realizing it and removing it helped us develop our simpler specification, especially in terms of control flows, which subsequently led us to easily discover the liveness violation in Replica.

Liveness violation in Replica and fix. Replica is arguably the most complex of the process types: it needs to mediate with both clients and leaders while competing with other replicas for proposed slots; perform eventual state updates in correct order of decided slots despite possibly receiving them out of order; and support reconfiguration.

Our high-level specification led us to discover a liveness violation in the original pseudocode: if no decision is received for a slot, e.g., due to lost propose messages from all replicas proposing for that slot, all replicas will stop applying decisions from that slot on, so slot_out will stop incrementing; furthermore, due to the limited window used for incrementing slot_in, all sending of proposals will stop after the window is used up. So all replicas will be completely stuck, and the entire system will stop making progress.

To fix, a replica can propose for that slot again after a timeout. A leader can then work on deciding for that slot if a decision for it has not been made; otherwise, it can send back the decision for that slot.

Liveness violation in Leader and fix. Leader is the core of Paxos. A leader must always be able to make progress unless it has crashed. However, liveness violations can occur in several ways.

For Phase 1, if after sending a 1a message, the leader does not receive 1b messages from a majority of acceptors or a preempt message, e.g., due to a lost 1a message, the leader will wait forever. To fix, the leader can add a timeout to the outer await, so a new round of Phase 1 will be started after the timeout.

For Phase 2, if for a newly proposed slot, no 2b messages are received from a majority, e.g., due to a lost 2b message, the leader will not make a decision for that slot. If this happens to all leaders, the replicas will not receive a decision for that slot, leading to the liveness violation in Replica described earlier. To fix, the leader can send the 2a message with that slot again after a timeout from waiting to receive a majority of 2b messages for it.

For Phase 2, if a preempt message is lost, and a majority has seen a larger ballot, the leader will fruitlessly continue to send 2a messages for newly proposed s, c and count received 2b message but not having a majority to make any decision, and thus not making progress. To fix, the leader can start a new round of Phase 1 after a timeout from waiting to receive a majority or a preemption.

5 OPTIMIZATIONS AND EXECUTIONS

High-level specifications can be too inefficient to execute, which is indeed the case with the specification in Figure 3 as well as any specification following the original pseudocode [44]. However, high-level specifications also allow additional optimizations and extensions to be done more easily. We specify the two most important ones suggested in [44] for vRA Multi-Paxos but not included in its pseudocode, describe a general method for merging processes that supports a range of additional optimizations, and discuss results of executions.

5.1 State reduction with maximum ballot

The most serious efficiency problem of the algorithm in Figure 3 is the fast growing set accepted in 1b messages, which quickly choking any execution of the algorithm that does real message passing. The solution is to not keep all triples in sent 2b messages, as in Figure 3 and the original pseudocode [44], but keep only triples with the maximum ballot for each slot, so there is at most one triple for each slot. This is done by changing line 44 to

\[
44 \quad \text{accepted} := \{(b,s,c): \text{sent ('2b',b,s,c),} \\
\quad \quad \quad \quad \quad \quad b = \max \{b: \text{sent ('2b',b,s,...)}\}\}
\]

This drastically reduces not only the size of 1b messages, but also the space needed by acceptors and leaders, which send and receive 1b messages, respectively.
5.2 Failure detection with ping-pong and timeout

Failure detection addresses the next most serious problem: leaders compete unnecessarily to become the leader with the highest ballot, leaving little or no time for proposals to be decided. Adding failure detection uses ping-pong after preemption: in Leader, after exiting the outer await following a preempt and before incrementing ballot, periodically ping the leader leader2 that has the larger ballot (r2, leader2) and wait for replies, by inserting

38.1 while each sent ('pong', r2, t) to =leader2
    has received ('pong', r2, t) from leader2:
38.2 send ('pong', r2, logical_time()) to leader2
38.3 await timeout TIMEOUT

and adding the following receive definition after the run definition:

39.1 receive ('pong', r2, t) from leader2:
39.2 send ('pong', r2, t) to leader2

TIMEOUT is a variable holding the timeout value in seconds.

5.3 Merging processes

High-level specifications in DistAlgo allow different types of processes that run at the same time to be merged easily, even if they interact with each other in sophisticated ways, provided they together have one main flow of control. There are two cases:

1. A process P that has only await false in run can be merged easily with any process Q. For example, in Figure 3, Acceptor can be merged with Leader by adding the receive definitions of Acceptor to the body of Leader.

2. A process P that has only while true: await... in run, with no timeout in the await, can be merged easily with any process Q. For example, in Figure 3, Replica can be merged with Leader by adding for each branch cond: stmt of await of Replica, a receive _ : if cond: stmt definition to the body of Leader.

Process setups can be transformed accordingly. Details are omitted because they are less important. These transformations are easy to automate. Inversely, independent receive definitions can be easily put into separate processes. In Figure 3, all three types of processes, or any two of them, can be merged, giving a total of 4 possible merged specifications.

Merging supports colocation of processes cleanly, and allows a range of optimizations, e.g., garbage collection of states of leaders and acceptors, for decided slots already learned by all replicas [44]. Furthermore, communication between processes that are merged no longer needs real message passing but can be done more efficiently through shared memory. Also, because the actions are independent, lightweight threads can be used to make each process more efficient.

With a few more small variations to vRA Multi-Paxos, merging Replica, Leader, and Acceptor into one process type yields essentially the 'Replica' in Chubby [7], Google’s distributed lock service that uses Paxos, and the 'Server' in Raft [40], a pseudocode for the main features of Chubby. In general, separate processes provide modularity, and merged processes reduce overhead. Being able to merge separate processes easily allows one to obtain the benefits of both.

5.4 Configuration and execution

A main definition, similar to that in Figure 2, can set up a number of Replica, Leader, and Acceptor processes, or their merged versions, and some Client processes that send request messages to Replica processes and receive response messages; and parameters WINDOW and TIMEOUT can be defined. We summarize the results of running DistAlgo specifications:

- DistAlgo specifications can be run directly. For example, a complete specification of vRA Multi-Paxos with state reduction and failure detection in file spec.da (available at darlab.cs.stonybrook.edu/paxos) can be run by executing pip install pytialgo followed by python -m da spec.da. The default is to run all processes on the local machine as separate operating system processes.
- The DistAlgo specification for vRA Multi-Paxos as in Figure 3, without the state reduction in Section 5.1, almost immediately overflows the default message buffer size of 4KB, yielding a MessageTooBigException.
- The DistAlgo specification for vRA Multi-Paxos with state reduction, without the failure detection in Section 5.2, runs continuously but most times stops making progress (decisions) for 3 leaders, 3 acceptors, and 3 replicas, serving 10 client requests, and was killed manually after 200 rounds (200–600 ballots) have been attempted.
- The DistAlgo specification for vRA Multi-Paxos with both state reduction and failure detection runs smoothly. For example, for 10 processes (3 leaders, 3 acceptors, 3 replicas, and 1 client), processing 10 requests takes 77.822 milliseconds (ranging from 74.141 to 84.825), averaged over 10 runs, on a Intel Core i7-6650U 2.20GHz CPU with 16 GB RAM, running Windows 10 and Python 3.7.0.

Additional optimizations. Many additional optimizations and experiments can be done, especially including transforming high-level queries into efficient incremental updates of auxiliary variables [34, 35], but they are beyond the scope of this paper. Note that incremental updates of auxiliary variables allow sent and received to become dead variables and be eliminated. Our experience is that precise high-level specifications allow us to understand the algorithms much better and to significantly improve both correctness and efficiency much more easily than was possible before.

6 CORRECTNESS AND FORMAL VERIFICATION

The problems of vRA Multi-Paxos described in Section 4 do not affect the safety of vRA Multi-Paxos. However, even safety is not easy to understand and needs verification.

We have developed formal proofs of safety of the complete specification of vRA Multi-Paxos in Figure 3, and of the one extended with state reduction and the one further extended with failure detection as described in Section 5. The safety property ensures that, for each slot, only a single command may be decided and it must be one of the commands proposed.

The proofs are done by first translating the specifications into TLA+, Lamport’s Temporal Logic of Actions [25]. This was done
systematically but manually. The high-level nature of our specifications makes the translation simple conceptually: each type of data in DistAlgo corresponds to a type of data in TLA+, and each expression and statement in DistAlgo corresponds to a conjunction of equations in TLA+. The three translated specifications of vRA Multi-Paxos and extensions in TLA+ are 154, 157, and 217 lines.

Automatic translators from DistAlgo to TLA+ had in fact been developed previously, and we are currently developing a new one. The earlier translators produced TLA+ specifications that contain many more details needed to handle general control flows, especially low-level flows, and are formidable for formal verification, even for simpler protocols. We are using experience from systematic manual translations of high-level specifications in building a new automatic translator.

6.1 Proofs in TLAPS

Our proofs are manually developed and automatically machine-checked using TLAPS [37], a proof system for TLA+. The proofs for the three translated specifications are 4959, 5005, and 7006 lines, respectively. The proofs are much more complex and longer than the previous proof of 1033 lines for Multi-Paxos with preemption [9], because of the additional details in vRA Multi-Paxos, and the extensions for state reduction and failure detection. Appendix B contains additional details about the proofs.

Compared to other manually developed and mechanically checked proofs of executable Multi-Paxos and variants, namely, a proof of safety and liveness of Multi-Paxos in Dafny from IronFleet [17] and a proof of safety of Raft [40] in Coq from Verdi [47], our proofs in TLAPS are 4 to 10 times smaller, compared with 30,000 lines in IronFleet and 50,000 lines in Verdi. We believe this is due to our use of high-level queries over message histories, which are much simpler and directly capture important invariants, in contrast to use of repeated and scattered lower-level updates, as studied for a more abstract specification [8]. Shorter proofs are much easier to understand and maintain, and also easier and faster to check automatically. Both are significant advantages for practical development cycles of specifications, programs, and proofs.

Our proof checking times are 9.5, 9.8, and 13 minutes, respectively, on an Intel i7-4720HQ 2.6GHz CPU with 16GB memory running Ubuntu16.04 LTS and TLAPS1.5.2. No proof checking time is reported for the proof from Verdi [47], but we were able to run proof check for the proof after solving some version mismatch problems, and it took 29 minutes to run on the same machine as our proof. The proof checking times for the proofs from IronFleet are reported to be 147 minutes for the protocol-level proof and 312 minutes including also the implementation-level proof (without specifying the machine used for the proofs) [17]; we have not been able to run proof check for their proofs on our machine due to an error from a build file.

6.2 Safety violation and fix

Our development of formal proofs also allowed us to discover and fix a safety violation in an earlier version of our specification for vRA Multi-Paxos. There, acceptors always reply with $1b$ and $2b$ messages, not $\text{preempt}$ messages, as in the original pseudocode, and leaders try to detect preemption. It is incorrect because the ballot number in leaders may increase after a $1a$ or $2a$ message is sent, contrasting the fixed ballot number in a Scout or Commander process used for detecting preemption. The safety violation was discovered after the proof could not succeed.

The fix of having acceptors detect preemption and inform the leader also makes the algorithm much more efficient in the case of preemption upon receiving $1a$ messages: a $\text{preempt}$ message with only a ballot number is sent, as in Figure 3, instead of a $1b$ message with a ballot number and an entire accepted set, as in the original pseudocode.

The earlier incorrect version was used in distributed algorithms and distributed systems courses for several years, with dozens of course projects and homeworks having used it, including ones directed specifically at testing and even modeling using TLA+ and model checking using TLC [36]. However, this safety violation was never found, because it requires delays of many messages, extremely unlikely to be found by testing or model checking, due to the large search space that must be explored.

7 RELATED WORK AND CONCLUSION

Consensus algorithms and variants around Paxos have seen a long series of studies, especially their specifications for understanding, implementation, and verification.

Paxos is well-known to be hard to understand. Since its initial description [23], much effort has been devoted to its better exposition and understanding. Earlier descriptions use English, e.g., [24], or state machines, e.g., [11, 28]. Later studies include pseudocode, e.g., [20, 40, 44], and deconstructed pseudocode or code, e.g., [6, 15, 45]. Among existing works, vRA Multi-Paxos pseudocode [44] is by far the most direct, complete, and concise specification of a more realistic version of Multi-Paxos. Our specification captures and improves over vRA Multi-Paxos pseudocode, and yet is much simpler and smaller, even though executable code is generally much larger and more complex than pseudocode.

In particular, our specification captures the control flows and synchronization conditions at a higher level than previous specifications, yet is precise, complete, and directly executable. It is exactly the higher-level, simpler specification that allowed us to easily discover the liveness violations in vRA Multi-Paxos if messages can be lost, and to find other issues and fixes. It has also helped tremendously in teaching [34].

An earlier work [33] uses similar high-level language constructs. However, it keeps the complex Leader process doing low-level updates while spawning Scout and Commander processes as in vRA Multi-Paxos. It does not have reconfiguration, state reduction, or failure detection; its more complex control flows make it more difficult to add them and be sure of correctness. Its Replica process uses a for loop and sequential statements in the loop body instead of high-level $\text{await}$ conditions, causing it to iterate extremely inefficiently, bias towards sending a proposal before applying decisions, and apply decisions exhaustively before sending more proposals. It also has the same liveness violations as in vRA Multi-Paxos but they were not discovered.

One might also try to understand Paxos variants through more practical implementations, but these implementations are much larger and more complex. For example, Google Chubby’s C++ server
code is reported to be about 7000 lines \[7, 10\] and is not open-source. Paxos for system builders \[20, 21\] is written in C and has over 5000 lines, not including over 2000 lines of library code for group communication primitives.\(^1\) OpenReplica \[1\] is implemented in Python and has about 3000 lines.\(^2\) They generally include many lower-level data structures, bookkeeping tasks, and language details. This makes it much harder to understand the algorithms used. They are also so far infeasible for formal verification, manually or with automated support.

There has been significant effort on formal specification and verification of Paxos and variants. Many specifications have been developed, especially for Basic Paxos, including earlier ones described previously \[35\]; our specifications are significantly simpler while including full details for execution in real distributed environments. Several proofs are successful. Some are automatic by writing specifications in a restricted language \[42, 43\]. Some include proofs of liveness properties, e.g., \[17, 41\]. Some also generate code, e.g., \[16, 43\]. Some proofs are discussed extensively but are only on paper \[15\]. Some others are also for abstract specifications that omit many algorithm details, e.g., \[5, 8, 9\]. Proofs for executable implementations are from IronFleet \[17\] and Verdi \[46\], with much larger proofs and longer proof checking times, as discussed in Section 6.

To the best of our knowledge, no previous efforts of specification and formal verification, for any Paxos variant, reported finding any correctness violations in published specifications or any improvements to them. However, Fonseca et al. \[14\] discovered 16 bugs in IronFleet, Verdi, and Chapar \[30\] for distributed key-value stores. These include bugs in protocol specification, verification tool, and shim layer modeling; no bugs were found in the protocols modeled. 11 of the 16 bugs were discovered by manual inspection, even without prior experience of Fonseca with OCaml, the language used by Coq.\(^3\) This helps support both the importance and difficulty of writing good specifications for understanding and manual inspection, as well as formal verification.

There are many directions for future research: higher-level specifications of more variants of Paxos and other important distributed algorithms, more powerful optimizations for automatically generating efficient implementations, and better methods for developing automated proofs directly from high-level specifications.

A VAN RENESSE AND ALTINBUKEN’S PSEUDOCODE FOR MULTI-PAXOS WITH PREEMPTION AND RECONFIGURATION

Figure 4 shows the Leader process that spawns Scout and Commander processes in vRA Multi-Paxos. Replica and Acceptor processes are not shown due to space limitations.

B MECHANICALLY CHECKED PROOFS IN TLAPS

We developed inductive proofs of safety for all three specifications. Like the proof for Multi-Paxos from \[9\], they are inductive proofs based on several invariants that together imply safety. The proofs involve three types of invariants: (1) type invariants, stating that as the system progresses, all data in the system have the expected types, (2) invariants about local data of processes, for example, about the values of ballot, accepted, and maxb, and (3) invariants about global data of the system, in particular, about the messages sent in the system.

Our proofs for vRA Multi-Paxos differ from the proof for Multi-Paxos from \[9\] for several reasons, including differences between the algorithms themselves. For example, the accepted set in vRA Multi-Paxos contains all triples for which a 2a message was sent and received and may contain a triple for which a 2b message was not sent, whereas in Multi-Paxos in \[9\], the accepted set would only keep a triple if a 2b message was sent containing that triple. Also, to keep our specification in TLA+ close to the specification in DistAlgo, we model ballots as tuples containing a natural number and a process ID, not as natural numbers in \[9\]. This modeling difference has huge impact on the proof, because comparison operators like \(>\) and \(\geq\) on natural numbers are built-ins in TLAPS, and are reasoned about automatically, but comparison operators on tuples need to be defined using predicates, and all of their properties, including fundamental properties like transitivity and non-commutativity, need to be explicitly stated in lemmas and proved. In addition, we specify and prove safety of three versions of Multi-Paxos, all of which are variations not considered in \[9\].

B.1 Results and comparisons

Figure 5 presents the results about our specifications and proofs of vRA Multi-Paxos and its extensions, and the specifications and proofs of Multi-Paxos from \[9\] and Basic Paxos from \[27\]. First, we compare the specifications and proofs of vRA Multi-Paxos and its extensions with each other:

- The specification size grows by only 3 lines (1.9%) from 154 when we add state reduction, but by 60 more lines (38%), for the new actions added, when we add failure detection.
- The proof size grows by only 46 lines (0.9%) from 4959 when we add state reduction, but by 2001 more lines (40%) when we add failure detection, roughly proportional to the increase in specification size.
- The maximum level and degree of proof tree nodes remain unchanged when state reduction is added. When failure detection is added, the maximum level of proof tree nodes remains unchanged, but the maximum degree of proof tree nodes increases by 20 (71%), from 28 to 48, due to more complex proofs for the new actions added for failure detection.
- An interesting decrease of one lemma is seen after state reduction is added. The lemma states that the maximum of a set is one of the maximums of its two partitions. This lemma was needed in the case when all triples in 2a messages are kept by the acceptors. However, owing to state reduction, only triples with the maximum ballots are kept, making the proofs simpler. The number of stability lemmas and their uses remain unchanged when we add extensions. A stability lemma is a lemma asserting that a predicate continues to hold (or not hold) as the system goes from one state to the next in a single step.
- The number of proofs by induction on set increment remains unchanged when we add extensions. The number of proofs by contradiction increases; in those cases, constructive proofs were more challenging.
- The number of obligations, i.e., conditions that TLAPS proves, increases by 153 (3.5%) from 4364 when state reduction is added, and by 1063 (24%) more when failure detection is added, contributing to the increase in proof size.
- The proof check time decreases by 21 seconds (3.6%) from 590 to 569 when state reduction is added. This was expected because, with state reduction, for each slot, only the triple with the maximum

\(^1\)From the “Download” link of “Paxos for System Builders” at http://wwwdsn.jhu.edu/software.html, April 15, 2018.
\(^2\)From email with Emin Gun Sirer, August 12, 2011.
\(^3\)From communication with Fonseca at and after a talk he gave, April 2018.
process Scout(\(\lambda\), acceptors, \(b\))
\[
\text{var } \text{waitfor} := \text{acceptors}, \text{pvalues} := \emptyset;
\forall \alpha \in \text{acceptors}: \text{send}(\alpha, (p1a, \text{self}(f), b));
\text{for ever}
\text{switch receive()}
\text{case } \langle p1b, \alpha, b', r \rangle:
\text{if } b' = b \text{ then}
\text{pvalues} := \text{pvalues} \cup r;
\text{waitfor} := \text{waitfor} \setminus \{\alpha\};
\text{if } |\text{waitfor}| < |\text{acceptors}|/2 \text{ then}
\text{send}(\lambda, (\text{adopted}, b, \text{pvalues}));
\text{exit();}
\text{end if}
\text{else}
\text{send}(\lambda, (\text{preempted}, b'));
\text{exit();}
\text{end if}
\text{end case}
\text{end switch}
\text{end for}
\text{end process}
\]

process Commander(\(\lambda\), acceptors, replicas, \((b, s, c)\))
\[
\text{var } \text{waitfor} := \text{acceptors};
\forall \alpha \in \text{acceptors}: \text{send}(\alpha, (p2a, \text{self}(f), (b, s, c)));
\text{for ever}
\text{switch receive()}
\text{case } \langle p2b, \alpha, b' \rangle:
\text{if } b' = b \text{ then}
\text{waitfor} := \text{waitfor} \setminus \{\alpha\};
\text{if } |\text{waitfor}| < |\text{acceptors}|/2 \text{ then}
\forall p \in \text{replicas}:
\text{send}(p, (\text{decision}, s, c));
\text{exit();}
\text{end if}
\text{else}
\text{send}(\lambda, (\text{preempted}, b'));
\text{exit();}
\text{end if}
\text{end case}
\text{end switch}
\text{end for}
\text{end process}
\]

process Leader(acceptors, replicas)
\[
\text{var } \text{ballot.num} := (0, \text{self}());, \text{active} := \text{false}, \text{proposals} := \emptyset;
\text{spawn}(\text{Scout}(\text{self}(), \text{acceptors}, \text{ballot.num}));
\text{for ever}
\text{switch receive()}
\text{case } \langle \text{propose}, s, c \rangle:
\text{if } \exists \alpha' : (s, c') \in \text{proposals} \text{ then}
\text{proposals} := \text{proposals} \cup \{(s, c)\};
\text{if } \text{active} \text{ then}
\text{spawn}(\text{Commander}(\text{self}()), \text{acceptors}, \text{replicas}, (\text{ballot.num}, s, c)));
\text{end if}
\text{end if}
\text{end case}
\text{case } \langle \text{adopted}, \text{ballot.num}, \text{pvals} \rangle:
\forall (s, c) \in \text{proposals}:
\text{spawn}(\text{Commander}(\text{self}()), \text{acceptors}, \text{replicas}, (\text{ballot.num}, s, c)));
\text{active} := \text{true};
\text{end case}
\text{case } \langle \text{preempted}, (r', \lambda') \rangle:
\text{if } (r', \lambda') > \text{ballot.num} \text{ then}
\text{active} := \text{false};
\text{ballot.num} := (r' + 1, \text{self}());
\text{spawn}(\text{Scout}(\text{self}(), \text{acceptors}, \text{ballot.num}));
\text{end if}
\text{end case}
\text{end switch}
\text{end for}
\text{end process}
\]

\[
pmax(\text{pvals}) \equiv \{(s, c) | \exists b : (b, s, c) \in \text{pvals} \land \forall b', c' : (b', s, c') \in \text{pvals} \Rightarrow b' \leq b\}
\]

\[
x \ast y \equiv \{(s, c) | (s, c) \in y \lor (s, c) \in x \land \exists b' : (s, c') \in y\}
\]

Figure 4: Leader with Scout and Commander processes in vRA Multi-Paxos pseudocode [44, Fig. 6, Fig. 7, and the two definitions on pages 12-14].
ballot is kept. Upon receiving a triple with a larger ballot, only the new triple is kept, and the maximum of a singleton set is the item itself, making the proof time decrease.

The proof check time increases by 212 seconds (37%) when failure detection is added. This is expected, because there are more proof obligations (24%) and the proof is larger (40%).

Next, we compare our TLA+ specification and proof of vRA Multi-Paxos (without state reduction or failure detection) with those of Multi-Paxos with preemption from [9].

- The specification of vRA Multi-Paxos, excluding comments, is 154 lines, which is 59% more. This increase is because [9] omits many algorithm details, while our specification models the many more details in Figure 3.
- The proof of vRA Multi-Paxos, excluding comments, is 4959 lines, which is 380% more. This increase is due to many factors, including more actions (for sending 2a messages in two cases and for sending decisions), more invariants (about the looser accepted set and about program points for the additional actions), and representing ballots as tuples instead of natural numbers, as mentioned above.
- The proof tree for vRA Multi-Paxos is more complex, as shown by the 65% increase, from 17 to 28, in the maximum degree of proof tree nodes and 9% increase, from 11 to 12, in the maximum level of proof tree nodes.
- Twice as many lemmas are needed for vRA Multi-Paxos, 24 vs. 12, because properties of operations on tuples need to be explicitly stated in lemmas and proved, as mentioned above.
- We prove 2 more stability lemmas for vRA Multi-Paxos for the additional actions. The number of uses of stability lemmas increases by 47 (162%), from 29 to 76, because of the additional actions and the larger number of invariants.
- The number of proofs by induction on set increment increases by 26 (650%), from 4 to 30, the number of proofs by contradiction increases from 1 to 14 (1300%), the number of obligations increases by 3405 (355%), from 959 to 4364, and the proof check time increases by 496 seconds (627%), from 94 seconds to 590 seconds, all due to increased complexity in the specification, more actions, and more invariants.

**Acknowledgements**

We thank Leslie Lamport and Robbert van Renesse for their clear explanations and helpful discussions about Paxos. We thank Bo Lin for his robust DistAlgo compiler with excellent support and his original DistAlgo program that follows vRA Multi-Paxos pseudocode directly. We thank hundreds of students in distributed algorithms and distributed systems courses and projects for extending, developing variants of, testing, evaluating, and model checking our executable specifications, including running them on distributed machines, in the cloud, etc. We thank Xuetian (Kain) Weng for developing earlier automatic translators from DistAlgo to TLA+. This work was supported in part by NSF under grants CCF-1414078, CCF-1248184, and CNS-1421893 and ONR under grant N000141512208.

**REFERENCES**


**Figure 5:** Comparison of results for safety proofs of Basic Paxos from [27], Multi-Paxos from [9], and vRA Multi-Paxos. Spec and proof sizes including comments are also compared because they are used in [9] as opposed to sizes excluding comments. Stability lemmas are called continuity lemmas in [9]. 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 Ubuntu 16.04 LTS and TLAPS 1.5.2.
We thank the reviewers for their helpful comments. We have revised the paper to address them as described below.

The revised paper is uploaded to EasyChair in the original submission part as a pdf file.

> *********************************
> 
> ## General Comments
> 
> > Our specification in DistAlgo is higher-level than the vRA Multi-Paxos pseudocode, in that it removed Scout and Commander processes and replaced repeated and scattered updates by high-level queries, making it almost completely declarative (Sec 3.2).
> > 
> > This makes the algorithm flow easy to see and allowed us to find the issues by just following it.
> > 
> > The vRA Multi-Paxos authors had implemented the algorithm in several languages, including Python (in appendix of the vRA paper) and previously Java, Erlang, and more before publishing the vRA paper. They did not find any of the issues we described.
> 
> This seems to be the main technical contribution of the paper: a demonstration on how previous work in high-level specification languages (DistAlgo) can be actually applied to identify real bugs in executable implementations. Making this clearer in the existing presentation would be beneficial for the reader. This could be addressed more directly in the Introduction of the paper -- this needs to be the more central idea with supporting items around a.) experimentally validating the specifications made in this language as correct with TLA; and b.) showing that proposed optimizations do not violate correctness. Adjustments should be carried throughout the sections accordingly.
> 
> Done.
> 
> Emphasized/added these points in intro, and later in the paper:

We found the issues by just following the simplified algorithm flows. (in intro and end of first paragraph of Sec. 4 on Issues and Fixes)

Previous implementations of vRA Multi-Paxos did not find the issues. (end of intro)

The safety of our specifications needs verification. (end of first paragraph of Sec. 6 on Verification, before saying we developed TLA+ specifications and TLAPS proofs for vRA Multi-Paxos and for the ones with optimizations)

> > Erlang does not have high-level queries, as you noted already, and it does not have await on conditions, so it cannot express complex synchronization conditions at a high level. Declarative queries for such conditions are exactly what make our specifications higher-level.
> 
> > There needs to be more background on the language that's being used -- preferably before getting into the Paxos specifications. It should at least be briefly mentioned what makes these things unique -- in the context of other work that's been presented at PPDP, like Erlang systems, etc. -- the authors should draw a distinction between the key features of DistAlgo that distinguish it from a language like Erlang that's been used for many implementations of complex distributed protocols. (e.g., Chord, Paxos, Raft, Viewstamped Replication Revisited.)
> 
> Done.
> 
> Added clarification that DistAlgo's constructs for high-level control flow and synchronization conditions are not in Erlang. (under last paragraph heading of Sec 2.2, end of the first paragraph)

> > Specific Comments
> 
> > * Can the specifications of Basic Paxos and Multi Paxos be combined somehow or reduced? There appears to be a bit of duplication between the sections and it might assist the reader. Possibly not.
> 
> Did not combine Lamport's paper on Paxos Made Simple is dedicated to Basic Paxos. We think it is important to make Basic Paxos clear first. It is also used extensively as examples for explaining the language before we get into Multi-Paxos and variants.

> > * The terminology of "almost completely declarative" seems rather imprecise. Is there another way to describe this? Is there more precise terminology that can be used? Maybe something like "declarative in <some aspect>?"
> 
> Done.
> 
> Added clarification that it means updating only a number for the protocol round besides the sets of messages sent and received. (in abstract, end of first paragraph, and in intro). It has already been explained in the paper body. (end of Sec. 3.2)
> * Please address the practicality of message histories that grow indefinitely in the text for the reader.

Done.

> * Is the purpose of the TLA translation to verify the specifications and optimizations you have made are sound? Is it used to demonstrate the applicability of the technique? Is it expected that users would translate and prove their algorithms with TLA as well? (If so, why not just write in TLA to begin with and then implement?)

The first as described above, added that our specifications and optimizations need verification. (end of first paragraph of Sec. 6)

> Please make every effort to address these.

> The final version of your paper is due to be submitted on the 15th July 2019 (AOE)

> We are looking forward to seeing you in Porto in October!

> Best wishes,

> PPDP'19 program committee

> SUBMISSION: 25
> TITLE: Moderately Complex Paxos Made Simple: High-level Executable Specification of Distributed Consensus Algorithms

> ----------- Overall evaluation -----------
> SCORE: 2 (accept)
> ----- TEXT:

> The paper presents variations of the Paxos algorithm for distributed consensus specified in a high-level executable language -- DistAlgo. The authors claim that the algorithms thus specified are precisely captured, are simpler and smaller than other representations. The specification of Multi-Paxos is used to show additional optimizations for state reduction and for failure detection. The resulting three specifications have been subjected to machine checked safety proofs, by being first translated to TLA+, and then checked using TLAPS. This has enabled the authors to discover and fix a safety violation in the initial representation and cut down useless replies, delays and liveness violations.

> I like the Paxos specifications in DistAlgo, they look much more clear and clean than other representations. Being executable is a clear plus. The paper could use some general polishing. There are some very long phrases that need breaking into separate sentences. The abstract is very difficult to read. The second paragraph, in particular, is hard to follow and needs refactoring. I would wish to see the versions of Paxos implemented mentioned here, together with the implementation language.

Done.

Long phrases are broken into separate sentences. (abstract, second paragraph).

Added Paxos version and language name. (abstract, first paragraph, second and third sentences)

> I found the sections on the language and on basic Paxos very clear. Section 3.1, on the other hand, was not very clear, as it goes into a lot of details before explaining the algorithm. For example, the notion of slots is used in the description of all processes, before being introduced. Perhaps some of section 3.3 could be moved to 3.1 to better explain the algorithm.

Added pseudocode for Leader process from vRA Multi-Paxos in Appendix A.

Slot is introduced (in italic) in the first paragraph of Sec. 3.

Did not move Sec. 3.3 to 3.1, because Sec. 3.3 is for our simplified spec. The original vRA Multi-Paxos is indeed not as easy to understand. (I myself had to ask/email/visit van Renesse for many questions).

> I would have liked to see at least some of the safety proofs in the paper. Otherwise, I find the representations and results convincing.

Not included due to space limitations.

The complete proof can be found at the URL given in the intro.

Our proofs for more abstract versions of Multi-Paxos can be found in our FM’16 and later papers.
This paper presents the design of a high-level executable specification language for distributed protocols. The main protocols examined in this paper are around consensus: specifically, the single-decree and multi-round variants of the Paxos consensus protocol. In this paper, the authors present an implementation of both of these protocols in their specification language, an evaluation based on bugs discovered in the protocols themselves, a comparison to the pseudocode specifications of these programs, and an evaluation of translating these implementations into TLA+ so that they can be verified using the TLAPS interactive proof assistant.

* Generally speaking, I quite enjoy what the paper purports to be -- I've always had my own suspicions that these extremely complex distributed protocols had undiscovered problems when turned into real systems. To that point, the authors do a good job of identifying a few of these issues and confirming them with the author that they are in fact, bugs. However, it seems like the authors only discovered these bugs through manual specification of these protocols in their language. It's unclear to me how the language that they describe helped discover these bugs -- and I think that's what's really lacking in the evaluation -- it seems like any implementer of these protocols could discover them if they were diligent in the testing of their software.

We found the issues by just following the simplified algorithm flows. (in intro and end of first paragraph of Sec. 4 on Issues and Fixes)

Previous implementations of vRA Multi-Paxos did not find the issues. (end of intro)

* On the topic of finding bugs in protocol implementations, one must look at the existing work in the field. Specifically, lineage-driven fault injection uses a high-level specification language and SMT to identify bugs from a formula extracted from an executable program -- granted, there exists no common, portable, or efficient runtime for the implementation language, but the work does exist. [The authors do cite the work on Bloom and it's implementation of the Synod algorithm -- but don't mention the work on LDFI.]

We looked at the LDFI paper in more detail, and found very minimum mention of Paxos. We referenced the Bloom implementation of the Synod algorithm, not the LDFI paper, because the focus of our paper is on complete specifications and proofs of Paxos, and the LDFI paper discusses no specifications or proofs of Paxos.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

We referenced the Bloom implementation of the Synod algorithm, not the LDFI paper, because the focus of our paper is on complete specifications and proofs of Paxos, and the LDFI paper discusses no specifications or proofs of Paxos.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.

* On the topic of high-level declarative protocol specification, I found it very difficult to differentiate DistAlgo from a language like Erlang. Erlang, for instance, has the ability to do blocking receives or async receives (via a gen_fsm or gen_server, etc.) supports multiple processes, control flow, setup, configuration, etc. I don't see why the declarative language of DistAlgo couldn't be expressed as Erlang (or for that matter, any actor language) with an extension for the declarative queries. In fact, I believe that using the standard libraries implementation of sets, along with Erlang's pattern matching behavior, all of this would be able to be built with minimal effort. The authors should differentiate the language from existing languages used in the implementation of distributed protocols in order to make the paper's contribution of a high-level executable specification language more compelling.
> * The presentation of Basic Paxos isn’t really used anywhere after it’s initial presentation -- with the exception of some references in
the Multi-Paxos section to explain it’s relationship to Basic Paxos. I found this problematic for a number of reasons: a.) it takes up
significant space to present and explain Basic Paxos, but it’s not really relevant once Multi-Paxos is presented; b.) Multi-Paxos
references a pseudocode algorithm that’s not included, but references the original paper where the pseudocode was presented making this
paper no longer self-contained (it was very confusing to see a reference to Figure 7 that I couldn’t find until I realized it was
Figure 7 *in a different paper.*) I would recommend that the authors investigate including more background on Multi-Paxos and focusing
on that protocol only because that’s where all of the contributions seem to be.

Addressed as described above:
Added pseudocode for Leader process from vRA Multi-Paxos (Figure 7 you mentioned) in Appendix A.
Also added reference to Appendix A immediately after the reference to Figure 7.

> * More generally: I really don’t understand what the paper is trying to be. The paper appears to initially sell a high-level
specification language with an implementation in Python with an information description of the language through use cases. Then, the
paper switches to the benefits of using the specification language to identify safety and liveness violations in protocols -- however,
the connection to the high-level specification language feels to the reader as anecdotal with phrases like “helped us discover the
problem more easily.” How easy? What does the programming model do to help the reader? Is it just being declarative? Are there
other declarative languages? This needs to be made more concrete. Is this paper an experience report on using a declarative language
in practice and how it helped the authors?

Addressed as described above:
Emphasized that it is the simplified algorithm flows that helped.

(DistAlgo is not purely declarative but contains constructs that
facilitate writing specifications and programs that are more declarative.
Our specifications are almost completely declarative as described above.)

> * Detailed Comments
> * Please call out the research questions you are asking early on -- it’s very difficult to understand what the authors have set out to
answer in this paper because it discusses many different things: a.) specification, b.) verification, c.) evaluation and performance.
The main question we ask is
"Can these algorithms be made completely precise, readily executable in real distributed environments, and at the same time easier to
understand?" (in intro, end of the first paragraph heading)

Also addressed as described above:
Emphasized/adDED these points in intro, and later in the paper:
We found the issues by just following the simplified algorithm flows.
(in intro and end of first paragraph of Sec. 4 on Issues and Fixes)
Previous implementations of vRA Multi-Paxos did not find the issues.
(end of intro)

> * The authors state the specifications -- which really just seem like programs/ implementations and not specifications -- are “almost
completely declarative” -- what does this mean? Can you be more precise?

Addressed as described above.

> * Section 2.3, paragraph 3 on the hardest parts is very dense. It should be rewritten to be clearer.

Done. Made all phrases more specific.

> * Section 3.1, references to previous work's Figures and pseudocode algorithms makes the work no longer self-contained.

> * Section 3.1, discusses simplifications of protocol -- as a reader, without having the other paper in front of me, I can’t tell if they
are simplifications. How can you evaluate or justify this? It seems anecdotal, in that it simplified the understanding for the
authors, but not necessarily other readers (e.g., Raft paper argumentation on why Raft is easier to understand for students.)

> * Section 3.2, 3.3, again, I’m left asking myself why we even discussed Basic Paxos at all at this point.

Addressed as described above:
Added the central part of the vRA Multi-Paxos pseudocode in Appendix A.

We will try to add more if there is space after putting in required ACM categories, etc.

> * Section 4, super neat that you found real bugs -- but, was this all just done manually? How did the tools help? How is it “easier”? 

Addressed as described above.
Section 5, confusing. You state that by adding these adjustments to the protocol it starts looking like Chubby -- OK, but Chubby came first, and didn’t it motivate the van Renesse work in 2015? (Wasn’t Chubby -- Paxos Made Live -- the first implementation of the algorithm you are investigating and the formal specification came after in 2015?)

This was answered in previous response to reviews.

Section 6 -- so wait, is the TLA specification extraction automated or not? It feels like not. The proof is manual as well. I’m confused, why is section here and how does it relate? If the extraction is automated, then you could argue that the specification language used help derive a more efficient algorithm to check with the proof assistant, but if the specification and proof are written manually, it’s unclear what this section is promoting.

This was answered in previous response to reviews.

Addressed as described above: Added that the safety of our specifications needs verification.

----------------------- REVIEW 3 ---------------------
SUBMISSION : 25
TITLE: Moderately Complex Paxos Made Simple: High-Level Executable Specification of Distributed Consensus Algorithms
AUTHORS: Yanhong A. Liu, Saksham Chand and Scott Stoller

----------- Overall evaluation -----------
SCORE: 2 (accept)
TEXT:
Summary:

This paper presents an interesting application of DistAlgo, a language for distributed algorithms with a formal operational semantics, and proposes a method for developing more concise specifications for both basic and more complex variants of the Paxos algorithm for distributed consensus. More specifically, it shows that DistAlgo can be used to give a natural description of the Paxos algorithms, and that the natural description can be used to improve the algorithms, resulting in a better algorithm specification than the published one. In addition, it explains that the resulting specifications in DistAlgo can be systematically translated into TLA+, and machine-checked safety proofs can be developed. This allows for detecting and fixing a subtle safety violation in an earlier specification.

Favorite Points

+ This paper demonstrates a very interesting application of DistAlgo. It cannot only naturally specify non-trivial Paxos algorithms, but also help to improve specification. In addition, with a translation from DistAlgo to TLA+, one can prove the safety property in a formal way.
+ It actually developed a new Paxos algorithm, which is better than the published one.
+ The paper is clearly presented and well structured. It is easy to follow and enjoyable to read.

Weak Points:

- Would it be possible to use other high-level calculus, such as CSP and CCS, for the same purpose? Some discussion on this would be helpful.

This was answered in previous response to reviews.

Added clarification that DistAlgo’s constructs for high-level control flow and synchronization conditions are not in CSP and CCS. (under last paragraph heading of Sec 2.2, end of the first paragraph)

----------------------- REVIEW 4 ---------------------
SUBMISSION : 25
TITLE: Moderately Complex Paxos Made Simple: High-Level Executable Specification of Distributed Consensus Algorithms
AUTHORS: Yanhong A. Liu, Saksham Chand and Scott Stoller

20
Generally, the idea of using a restricted set of a high-level implementation language such as Python to specify an executable version of a distributed system is appealing. See e.g. DSLabs which is popular among students to teach distributed systems at the University of Washington. Beschastnikh’s PGo follows a related idea for the Go implementation language.

Thanks again!

The major contribution of DistAlgo is the ability to a) concisely and b) formally specify an c) executable distributed system.

- Using lines of code is a very rough metric to capture complexity and readability. One can compress code by an order of magnitude, but it has quite the opposite effect on readability. The paper also seems to compare apples and oranges (section 7): The DistAlgo spec of Paxos is short but the paper later admits that a real-world implementation requires several optimizations. It would have been more realistic to compare the spec with those optimizations. Related, the paper does not indicate if DistAlgo supports some sort of refinement. It would also be interesting for the paper to compare the DistAlgo spec with a corresponding TLA+ spec (not the DistAlgo to TLA+ translation though).

Removed mention of lines of code in Related work section.

The others were answered in previous response to reviews.

- The oddity is that one has to _manually_ translate the implementation to TLA+ to formally verify the specification. Given the similarities of the DistAlgo language with PlusCal and the claim that it is straightforward to forward to translate from DistAlgo to TLA+, I wonder why they did not choose to go the other way around (derive DistAlgo code from a TLA+ spec with the major work being done in TLA+).

- The paper omits discussion of the process to translate counter-examples obtained from the TLA+ specification back to the original DistAlgo spec. In other words - assuming a proper DistAlgo to TLA+ translator - how will the reverse "conversion" work? The paper is pathological in the sense that it studies a well-known algorithm which is specified top to bottom. In this scenario it is fine to translate to TLA+ once. A newly developed algorithm will however require several iterations.

This was answered in previous response to reviews.

- The paper only briefly talks about the general problem of continuously growing message histories (section 5.1) yet message histories are considered a major feature of DistAlgo. It appears as if users are expect to find ways to manually truncate message histories. This seems like a source of correctness bugs.

This was answered in previous response to reviews.

Addressed as described above:

Added that incremental updates allow them to become dead variables and be removed. (middle of last paragraph of Sec. 5)