A Formal Treatment of Lamport's Paxos Algorithm *

Roberto De Prisco † Nancy Lynch‡ Alex Shvartsman§ Nicole Immorlica ¶

Toh Ne Win ||

October 10, 2002

Abstract

This paper gives a formal presentation of Lamport's Paxos algorithm for distributed consensus. The presentation includes a state machine model for the complete protocol, a correctness proof, and a performance analysis. The correctness proof, which shows the safety properties of agreement and validity, is based on a mapping to an abstract state machine representing a non-distributed version of the protocol. The performance analysis proves latency bounds, conditioned on certain timing and failure assumptions. Liveness and fault-tolerance correctness properties are corollaries of the performance properties.

*This work was supported in part by the NSF ITR Grant CCR-0121277.

†

‡Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA. Email: `lynch@theory.lcs.mit.edu`.

§Department of Computer Science and Engineering, 191 Auditorium Road, Unit 3155, University of Connecticut, Storrs, CT 06269 and Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-316, Cambridge, MA 02139, USA. Email: `alex@theory.lcs.mit.edu`.

¶Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-???, Cambridge, MA 02139, USA. Email: `nickle@theory.lcs.mit.edu`.

||Massachusetts Institute of Technology, Laboratory for Computer Science, 200 Technology Square, NE43-413, Cambridge, MA 02139, USA. Email: `tohn@mit.edu`.

1 Introduction

Overview: Lamport's Paxos algorithm [1] has at its core a solution to the problem of distributed consensus. In that consensus algorithm, the safety properties—agreement and validity—hold in all executions. The liveness property—termination—holds in all executions in which timing and failure behavior stabilize from some point onward. (The impossibility result of Fischer et al. [2] implies that the algorithm cannot be guaranteed to terminate under arbitrary timing and failure assumptions.) The paper of Dwork, Lynch, and Stockmeyer [3] presents another algorithm with this combination of guarantees.

The Paxos consensus algorithm is interesting both from a theoretical and a practical point of view. It can be used to build other algorithms: For example, Lamport [1] shows that it can be used to implement a totally ordered broadcast service; such a service can be used, in turn, to implement replicated state machines, by broadcasting messages representing update requests. The Paxos consensus protocol is also used to implement the reconfiguration part of the Rambo reconfigurable atomic memory algorithm [4, 5].

The Paxos algorithm has a long history: The algorithm was presented first in a 1988 technical report [6], which is essentially identical to [1]. Lampson's course notes [7] and expository paper [8] explained the technical ideas of [6] in other terminology and identified some key properties of the protocol.

De Prisco's M.S. thesis [9] and subsequent publications based on that thesis [8, 10] gave a formal presentation of the Paxos algorithm, following [6] closely. Specifically, those papers include a formal specification of the algorithm in terms of timed automata, and complete proofs of safety and latency properties, all done by hand. De Prisco's presentation represents the algorithm as a composition of a "main algorithm" and three timing-dependent components that together determine when it is time to start a new ballot. The main algorithm performs several tasks; one (the dissemination of final decisions) is timing-dependent, while the others are asynchronous.

Guerraoui's paper on "Deconstructing Paxos" [11] describes a way of decomposing the Paxos algorithm using a "round-based register" abstraction, which encapsulates the handling of the quorums. Later papers on Disk Paxos [12, 13] present variations on the Paxos protocol that work in settings in which the algorithm's state is maintained on separate disks. Lampson's invited talk at PODC 2001 [14] provides a unified description of basic Paxos, Disk Paxos and other related algorithms, using state machines at several levels of abstraction. Finally, Lamport has written a recent note [15] containing an informal explanation of the original Paxos algorithm.

This paper contains a new formal presentation of the original Paxos consensus algorithm [1], a proof of correctness, done by hand and checked using an interactive theorem-prover, and a latency analysis.

The Paxos consensus algorithm: The Paxos algorithm works roughly as follows: a leader starts "ballots", tries to associate a value to each ballot, and tries to collect enough approval for each ballot to use the value of that ballot as the decision value. The leader bases its choice of a value to associate with a ballot on the information returned by a quorum of the processes. Once the value is associated with the ballot, the leader tries to collect approval from a quorum of processes; if it succeeds in doing this, the ballot's value becomes the final consensus decision value.

In a bit more detail, the communication pattern followed by the traditional algorithm is as follows:

1. The leader starts a ballot and tells other processes about it. Any process that hears about the new ballot abstains from all earlier ballots for which it has not already voted.

2. Any process that has heard about the new ballot sends the value of the latest ballot it has voted for (if any) to the leader. When the leader hears from a quorum, it chooses a value for the new ballot based on values it has heard from the quorum.

3. The leader tells the other processes about the ballot's value. Any process that hears that a value has been chosen for the ballot, and that has not already abstained from the ballot, may vote for the ballot.

4. Any participant that has voted for the ballot tells the leader (and the other processes) about the vote. When the leader (or any other process) hears that a quorum of processes have voted for the ballot, it decides on that ballot's value.

In general, several leaders may operate at the same time, and may interfere with each other's work. However, in the "normal case", when failures stop and timing of events is "reasonable", only one leader will operate, and will ensure that a ballot completes.

The leader is chosen using a separate leader-election service. The leader-election service may, for example, elect the process with the highest id among those that seem to be operating. The service uses timeouts to determine which processes are currently operating.

The current leader decides when it should start new ballots, based on timing out old ballots. This arrangement means that the algorithm has two timing-dependent parts: the mechanism within the leader-election service that determines who has failed, and the mechanism within the main algorithm that determines when a leader should start a new ballot.

Our presentation: Our new presentation divides the algorithm into two components: a main algorithm and a $BallotTrigger$, which is responsible for saying when each participant should initiate a new ballot. The $BallotTrigger$ includes all the timing-dependent aspects of the protocol; the main algorithm is purely asynchronous.

The correctness properties (agreement and validity) do not depend on timing; therefore, our proof of these properties is based on the main algorithm alone, and not the $BallotTrigger$. The proof of agreement and validity uses state machines at four levels of abstraction. The top-level machine, $Cons$, is simply a representation of the consensus problem specification. The next level, $Global1$, is a high-level "global" description of the behavior of the algorithm. Although this level specifies that invocations to and responses from the service occur at particular locations, it does not specify locations for certain other key activities in the protocol, such as the creation of a ballot or the association of values with ballots. The third level, $Global2$, is a minor modification of $Global1$. It represents an optimization that brings the algorithm more in line with Lamport's version. Finally, the lowest level represents a complete distributed protocol. The two global state machines are similar to state machines that have been used by Lampson [14] and Lynch [16]. The correctness proof uses forward simulations between successive levels. The proof has been done by hand and also checked using the Larch interactive theorem-prover [17].

The performance analysis proves latency bounds, conditioned on certain timing and failure assumptions. To present this analysis, we introduce the $BallotTrigger$ component and add some constraints to the main Paxos algorithm. Some of these constraints are timing constraints, for example, we specify when the components must send messages and when they must take steps. Some are non-timing constraints, for example, we restrict the contents of messages in some places where the abstract protocol allows nondeterminism. We also introduce assumptions on message delay. Latency proofs are based formally on a "free timed automaton" model, which is based on the Clock Timed Automaton model introduced by DePrisco [9]. This is a formal device that allows us to describe timing assumptions that hold during some portions of an execution, but not necessarily throughout the entire execution.

This presentation is intended to help people in designing algorithms that use consensus as a building block, and in analyzing the correctness and performance of those algorithms precisely, using corresponding claims about the consensus service. In fact, we developed this presentation of Paxos for precisely this purpose: we needed such a consensus service as a building block in our work on the RAMBOreconfigurable atomic memory algorithm [4, 5]. RAMBOuses a sequence of Paxos consensus algorithms, one to agree on each successive configuration in the sequence of configurations. The latency bounds for Paxos decisions, presented here, are used in RAMBOto prove latency bounds for read, write, reconfiguration operations, under certain particular fault-tolerance and timing assumptions.

Related work: This work is most closely related to the earlier work of DePrisco [9, 8, 10]. The main differences are:

1. Our presentation uses state machines at four levels of abstraction; De Prisco's presentation uses only one level, representing the complete algorithm.

2. Our description of the main algorithm keeps less protocol state, for example, we do not explicitly keep track of where a process is in a particular phase of a particular ballot. Instead, we maintain a collection of information about the progress of ballots, in a form that allows it to grow monotonically over time. Instead of strict phase-oriented communication, we allow our processes to "gossip" this information in the background, at any time. Gossiped information is absorbed into the state of a recipient process, thus increasing the recipient's collection of information. When a process' local information satisfies certain conditions, the process is able to advance the protocol, for example, to associate a value with a ballot or to determine a final decision value. This structure simplifies the models and the proofs. (Lampson [14] has also advocated this approach.)

3. Our lowest-level algorithm description uses a somewhat different component decomposition from De Prisco's: we combine three of his components into one, the $BallotTrigger$. In our presentation, this is the only component that is timing-dependent. However, we do have to make other timing assumptions about our algorithm and its environment (restrictions on the gossip pattern, message delay bounds, and local processing time bounds) in order to prove the latency bounds.

4. We omit the explicit mechanisms by which the leader notifies other participants about decisions. Our gossiping pattern makes this mechanism unnecessary, because the gossiped information enables all participants to decide independently.

5. Our safety proofs are more formal, to the extent of using a theorem-prover.

Paper organization. This paper is organized as follows. Section 2 contains mathematical preliminaries. Section 3 contains the definition of the consensus problem, in terms of traces. Section 4 contains our three "global" specifications, $Cons$, $Global1$ and $Global2$, and the proofs that $Global1$ implements $Cons$ and that $Global2$ implements $Global1$. Section 5 contains the main part of the distributed Paxos algorithm, which is completely asynchronous. Section 6 contains our models for timing-dependent aspects of the algorithm.

2 Mathematical Foundations

2.1 Set Theory

We will define a number of basic data types. We assume a distinguished element $\perp$, which is not in any of the basic types. For any type A, we define a new type $A_\perp = A \cup \{\perp\}$. If A is a partially ordered set, we augment its ordering by assuming that $\perp < a$ for every $a \in A$.

2.2 I/O Automata (IOAs)

We use I/O automata as in [18], without tasks. These include states, start states, signature (input, output, internal), and transitions. We assume input-enabling.

2.3 General Timed Automata (GTAs)

These are also defined in [18]. If $\mathcal{A}$ is an I/O automaton, we define $Timed(\mathcal{A})$ to be the GTA which is just like $\mathcal{A}$ except that we add in all $(s, \nu(t), s)$ steps for all states s and all $t > 0$.

2.4 Free Version of a General Timed Automaton

If $\mathcal{A}$ is a GTA, then we define a variant of $\mathcal{A}$, which we call $\mathit{free}(\mathcal{A})$. This variant behaves like $\mathcal{A}$, except that it relaxes time constraints by allowing any amount of time to pass in situations where $\mathcal{A}$ specifies that a particular amount of time should pass. Our notion is similar to the notion of a Clock GTA introduced by DePrisco [9]. However, the new version is formulated in a more abstract style. Our definition includes technicalities to handle the special case of zero time passage.

More precisely, we define $\mathit{free}(\mathcal{A})$ to be the same as $\mathcal{A}$, except that:

1. We add a new, special internal action $\nu(0)_{\mathcal{A}}$, and add all steps of the form $(s, \nu(0)_{\mathcal{A}}, s')$ for which $(\exists t \in R^+)[(s, \nu(t), s') \in \mathit{trans}(\mathcal{A})]$.
2. The time-passage transitions are exactly those $(s, \nu(t'), s')$ such that either $s = s'$ or $(\exists t \in R^+)[(s, \nu(t), s') \in \mathit{trans}(\mathcal{A})]$.

Note that $\nu(0)_{\mathcal{A}}$ is a discrete action, not a special case of the distinguished time-passage actions $\nu(t)$, $t > 0$. The subscript $\mathcal{A}$ is to indicate that this action is special to the particular automaton $\mathcal{A}$.

The next definition formalizes the notion that an execution satisfies its timing constraints from some point onward.

Let $\mathcal{A}$ be a GTA. Let α be an execution of $\mathit{free}(\mathcal{A})$, and α' a finite prefix of α. Then we say that α is *regular after* α' provided that all steps that occur in α after α' are steps of $\mathcal{A}$. Again, this definition borrows from one in [9, 10].

3 Consensus Safety Specification

In this section, we define the problem of distributed consensus.

3.1 Data Types

In order to define consensus, we need to introduce some data types. The processes, or nodes, themselves are specified by their identifiers, elements of a totally-ordered set: Thus, we assume:

- I, the totally-ordered set of *endpoints* (a finite set).

The values that the nodes can agree upon are drawn from a universe of possible values:

- V, the set of possible *consensus values*.

3.2 Trace Property

In the consensus problem, participants in I may submit values in V. The consensus service is allowed to return decisions to processes that have submitted values. All decisions must be the same, and must be equal to some submitted value. In this section, we consider only safety aspects of the problem, not liveness aspects.

We formalize this problem as a *trace property*, which is an external signature and a set of traces over that signature. The signature appears in Figure 1.

Input:	Output:
$\mathsf{init}(v,i)$, $v \in V$, $i \in I$ $\mathsf{fail}(i)$, $i \in I$	$\mathsf{decide}(v,i)$, $v \in V$, $i \in I$

Figure 1: $Cons$: External signature

We use two auxiliary definitions—of safety assumptions and safety guarantees—to define the set of traces. The safety assumptions express constraints on the environment. In particular, they express the assumption that initiation occurs only once at each location, and a failed location does not initiate. The safety guarantees describe the properties we would like a distributed consensus algorithm to have. We ask that the behavior be well-formed, that is, only working (not failed) initiated nodes output a decided value, and they do this at most once. We also require the behavior exhibit agreement (all nodes that output a value output the same value) and validity (the decided value is a proposed one). Both the safety assumptions and the safety guarentees are defined formally as properties of a sequence β of actions in the external signature.

Safety assumptions:

- *Well-formedness:* For any $i \in I$:
  - At most one $\mathsf{init}(*, i)$ event occurs in β.
  - No $\mathsf{init}(*, i)$ event is preceded by a $\mathsf{fail}(i)$ event.

Safety guarantees:

- *Well-formedness:* For any $i \in I$:
  - If a $\mathsf{decide}(*, i)$ event occurs in β, then it is preceded by an $\mathsf{init}(*, i)$ event.
  - No $\mathsf{decide}(*, i)$ event is preceded by a $\mathsf{fail}(i)$ event.
  - At most one $\mathsf{decide}(*, i)$ event occurs in β.
- *Agreement:* If $\mathsf{decide}(v, i)$ and $\mathsf{decide}(v', i')$ occur in β, then $v = v'$.
- *Validity:* If a $\mathsf{decide}(v, i)$ event occurs in β, then it is preceded by an $\mathsf{init}(v, *)$.

The set of traces comprising the safety part of the consensus specification is exactly the set of traces that satisfy the implication: safety assumptions imply safety guarantees. Thus, if the environment fails to satisfy the safety assumptions, the resulting behavior of the consensus algorithm is unconstrained and the resulting trace is still a valid solution to the consensus problem. However, our main interest is in the behavior of the algorithm when the safety assumptions are satisfied, and so we will force our automata to recognize initialization only of a previously uninitialized and working (not failed) node.

4 Untimed Global Safety Specifications

In this section, we give a simple state machine, $Cons$, that solves the consensus problem defined in Section 3, and two successive refinements, $Global1$ and $Global2$, of $Cons$, which are non-distributed ("global") versions of the Paxos algorithm. $Cons$ can be regarded as an alternative specification of the consensus problem. $Global1$ and $Global2$ are useful as intermediate specifications, to show that the distributed versions of Paxos that we present in Sections 5 and 6 are correct.

The state machines at these two levels are derived from [16] which borrows ideas liberally from [14] and earlier proofs of Paxos.

The formal Larch Prover proof of all invariants and forward simulations presented in this section can be found in Appendix **??**.

4.1 Cons

The consensus problem is captured by the traces of the specification automaton, $Cons$. A state machine description of $Cons$ is given in Figure 2.

Signature:

Input:
  $\mathsf{init}(v, i), v \in V, i \in I$
  $\mathsf{fail}(i), i \in I$

Output:
  $\mathsf{decide}(v, i), v \in V, i \in I$

Internal:
  $\mathsf{choose\text{-}val}(v), v \in V$

State:
$initiated \subseteq I$, initially $\emptyset$
$proposed \subseteq V$, initially $\emptyset$
$chosen \subseteq V$, initially $\emptyset$
$decided \subseteq I$, initially $\emptyset$
$failed \subseteq I$, initially $\emptyset$

Transitions:

Input $\mathsf{init}(v, i)$
Effect:
  if $i \notin initiated \land i \notin failed$ then
    $initiated \leftarrow initiated \cup \{i\}$
    $proposed \leftarrow proposed \cup \{v\}$

Internal $\mathsf{choose\text{-}val}(v)$
Precondition:
  $v \in proposed$
  $chosen = \emptyset$
Effect:
  $chosen \leftarrow \{v\}$

Output $\mathsf{decide}(v, i)$
Precondition:
  $i \notin failed$
  $i \in initiated - decided$
  $v \in chosen$
Effect:
  $decided \leftarrow decided \cup \{i\}$

Input $\mathsf{fail}(i)$
Effect:
  $failed \leftarrow failed \cup \{i\}$

Figure 2: $Cons$ automaton

We claim that $Cons$ is a specification automaton for the consensus problem:

Lemma 4.1 *If β is a trace of Cons that satisfies the safety assumptions, then β satisfies the safety guarantees.*

Proof. □

If we can find a forward simulation from $Paxos$ to $Cons$, Lemma 4.1 implies that $Paxos$ solves the consensus problem. We achieve this goal by defining two successive refinements of $Cons$, $Global1$ and $Global2$, and showing forward simulations from $Global1$ to $Cons$, $Global2$ to $Global1$, and $Paxos$ to $Global2$.

4.2 Global1

Our first refinement is called *Global1*, includes most of the interesting ideas of the final distributed automaton, $Paxos$. The automaton description appears in Figure 3.

The $Global1$ automaton introduces the notion of ballots. A Ballot is simply a tuple consisting of an identifier and a (possibly nil) value from the set of consensus values V. Ballot identifiers are totally ordered. We assume:

- BId, a totally ordered set of *ballot identifiers.*

$Global1$ also introduces internal $vote$ and $abstain$ actions, by which participants assent to or promise not to assent to a ballot. As in the final algorithm, $Global1$ determines the fate of a ballot by considering the actions of *quorums* on that ballot.

We use these data types, for the rest of the paper.

- $read\text{-}quorums$, a set of finite subsets of I.
- $write\text{-}quorums$, a set of finite subsets of I.

We assume the following constraint: For every $R \in read\text{-}quorums$ and every $W \in write\text{-}quorums$, $R \cap W \neq \emptyset$.

Both $Paxos$ and $Global1$ consider a ballot "dead" if every node in a read quorum has abstained from it, and allows a ballot to succeed only if every node in a write quorum has voted for it.

There are several major differences between $Global1$ and $Paxos$. $Global1$ is not a composition of automata, that is, it is not a distributed algorithm. Although nodes can $abstain$ and $vote$ on ballots, the creation of ballots and the assignment of values to ballots are still global actions. Furthermore, the value assignment action does not fully capture the corresponding $Paxos$ automaton action: Instead of just checking the largest ballot $b' < b$ from which a read quorum has not abstained, it guarantees agreement by checking that *every* ballot $b' < b$ is either dead or has value v.

In order to show that $Global1$ satisfies the safety requirements for consensus, we map it to $Cons$ using a forward simulation. The mapping is the identity on all the state variables of $Cons$ except for $chosen$:

$$chosen = \{val(b) : b \in succeeded\}.$$

The forward simulation definition is, for the most part, straightforward. When an external action is fired in $Global1$, the corresponding external action is fired in $Cons$. Intuitively, having a write quorum vote for a ballot in $Global1$ corresponds to the $Cons$ automaton choosing the value of that ballot. Therefore, the $internalDecide$ action of $Global1$ should correspond to the $chooseVal$ action of $Cons$. However, the $Global1$ automaton can internally decide on ballots multiple times while the $Cons$ automaton can only choose a value once. Furthermore, the $Global1$ automaton can internally decide on a ballot that has not yet been assigned a value. Therefore we cannot blindly fire the $chooseVal$ transition in $Cons$ whenever the $Global1$ automaton performs an $internalDecide$ action. Also, we cannot simply map $assignVal$ actions to the empty sequence. Instead, we must carefully analyze the state of the automaton just before an $internalDecide(b)$ or $assignVal(b, v)$ action to decide whether to map the action to $chooseVal$ or the empty sequence. The trickiest part of the proof arises when $chooseVal$ has already been fired, and we try to map one of the above actions to the empty sequence. In this case, we are either adding ballots to the succeeded set, or possibly assigning values to ballots in the succeeded set. However, we are not changing the chosen set, and yet our mapping claims the succeeded set and the chosen set are essentially the same.

To preserve our mapping, we need some invariants. There is a key invariant that solves the problem. Namely, all ballots and values that we are adding to the succeeded set must agree with the values of the ballots already in the succeeded set. In fact, the value of *any* non-nil ballot must agree with the value of all live ballots with a smaller identifier *whether or not these ballots have been made*:

$$\forall b, b' \in BId,\ (b \neq \perp \wedge b' < b) \Rightarrow (val(b') = val(b) \vee b' \in dead)$$

Signature:

Inputs:
  $\mathsf{init}(v, i)$, $v \in V$, $i \in I$
  $\mathsf{fail}(i)$, $i \in I$

Outputs:
  $\mathsf{decide}(v, i)$, $v \in V$, $i \in I$

Internal:
  $\mathsf{make\text{-}ballot}(b)$, $b \in BId$
  $\mathsf{abstain}(B, i)$, $B \subseteq BId$, $B \neq \emptyset$, $i \in I$
  $\mathsf{assign\text{-}val}(b, v)$, $b \in BId$, $v \in V$
  $\mathsf{vote}(b, i)$, $b \in BId$, $i \in I$
  $\mathsf{internal\text{-}decide}(b)$, $b \in BId$

State:

$initiated \subseteq I$, initially $\emptyset$
$proposed \subseteq V$, initially $\emptyset$
$decided \subseteq I$, initially $\emptyset$
$failed \subseteq I$, initially $\emptyset$
$ballots \subseteq BId$, initially $\emptyset$
$succeeded \subseteq BId$, initially $\emptyset$
$val \in BId \to V_\perp$, initially everywhere $\perp$
$voted \in I \to 2^{BId}$, initially everywhere $\emptyset$
$abstained \in I \to 2^{BId}$, initially everywhere $\emptyset$

Derived variables:

$dead \subseteq BId$, defined as $\{b : (\exists R \in \mathit{read\text{-}quorums})(\forall j \in R)[b \in abstained(j)]\}$.

Transitions:

Input $\mathsf{init}(v, i)$
Effect:
  if $i \notin initiated \wedge i \notin failed$ then
    $initiated \leftarrow initiated \cup \{i\}$
    $proposed \leftarrow proposed \cup \{v\}$

Internal $\mathsf{make\text{-}ballot}(b)$
Precondition:
  $b \notin ballots$
Effect:
  $ballots \leftarrow ballots \cup \{b\}$

Internal $\mathsf{abstain}(B, i)$
Precondition:
  $i \notin failed$
  $i \in initiated$
  $voted(i) \cap B = \emptyset$
Effect:
  $abstained(i) \leftarrow abstained(i) \cup B$

Internal $\mathsf{assign\text{-}val}(b, v)$
Precondition:
  $b \in ballots$
  $val(b) = \perp$
  $v \in proposed$
  $(\forall b' \in BId, b' < b)[val(b') = v \vee b' \in dead]$
Effect:
  $val(b) \leftarrow v$

Internal $\mathsf{vote}(b, i)$
Precondition:
  $i \notin failed$
  $i \in initiated$
  $b \in ballots - abstained(i)$
Effect:
  $voted(i) \leftarrow voted(i) \cup \{b\}$

Internal $\mathsf{internal\text{-}decide}(b)$
Precondition:
  $b \in ballots$
  $(\exists W \in \mathit{write\text{-}quorums})(\forall j \in W)[b \in voted(j)]$
Effect:
  $succeeded \leftarrow succeeded \cup \{b\}$

Output $\mathsf{decide}(v, i)$
Precondition:
  $i \notin failed$
  $i \in initiated - decided$
  $b \in succeeded$
  $v = val(b)$
Effect:
  $decided \leftarrow decided \cup \{i\}$

Figure 3: *Global1* automaton

Some other more technical invariants are also necessary. A set of invariants sufficient to prove the forward simulation are:

1. $\forall i \in I\ voted(i) \cap abstained(i) = \emptyset$.
   That is, a node cannot vote for and abstain from the same ballot.

2. $\forall b \in BId\ val(b) \neq \bot \Rightarrow val(b) \in proposed$.
   That is, assigned values were proposed at some point.

3. $succeeded \cap dead = \emptyset$.
   No ballot is both succeeded and dead.

4. (The key invariant:) $\forall b, b' \in BId,\ (b \neq \bot \wedge b' < b) \Rightarrow (val(b') = val(b) \vee b' \in dead)$.
   A non-nil ballot's value agrees with the value of all live ballots less than it.

5. $succeeded \subseteq ballots$.
   Succeeded ballots were made at some point.

6. $\forall b \in BId\ b \in succeeded \Rightarrow (\exists W \in write\text{-}quorums\ \forall i \in W\ b \in voted(i))$.
   Every succeeded ballot has a write quorum that voted for it.

7. $\forall b \in BId\ b \notin ballots \Rightarrow val(b) = \bot$.
   If a ballot wasn't made, its value is $\bot$.

These should be easy to prove. The main interesting step seems to be the $internal\text{-}decide(b)$, in the case where $succeeded \neq \emptyset$ in the pre-state. In this case, we want the step to simulate a trivial (0-step) execution fragment in $Cons$. The key fact is that $val(b)$ is the same as $val(b')$ for any $b' \in succeeded$. This should follow from the invariants above (neither ballot is dead).

4.3 Global2

The second refinement is a forward simulation from an automaton called $Global2$ to the $Global1$ automaton. The $Global2$ automaton is exactly the same as the $Global1$ automaton except in the $\mathsf{assignVal}(b, v)$ action. This transition mimics the corresponding Paxos action exactly; it only checks that the largest ballot $b' < b$ from which a read quorum has not abstained has value v if such a b' exists:

Internal $\mathsf{assign\text{-}val}(b, v)$
Precondition:
  $b \in ballots$
  $val(b) = \bot$
  $v \in proposed$
  $(\forall b' < b)[b' \in dead]$
   $\vee(\exists b' < b)[val(b') = v \wedge (\forall b'', b' < b'' < b)[b'' \in dead]]$
Effect:
  $val(b) \leftarrow v$

It is straightforward to show that $Global2$ implements $Global1$, using a forward simulation. Because the state variables of $Global1$ and $Global2$ are the same, we can use the identity mapping. The only non-trival part of the forward simulation is the $\mathsf{assignVal}$ transition, where we have to show that the new precondition implies the corresponding old one. We do this by proving the "key invariant" holds in $Global2$:

Lemma 4.2 *The "key invariant":*

$$\forall b, b' \in BId,\ (b \neq \bot \wedge b' < b) \Rightarrow (val(b') = val(b) \vee b' \in dead)$$

holds in $Global2$.

Proof. Clearly, this invariant holds at the start. We must check this invariant continues to hold as ballots die and get assigned values. When a ballot dies, the statement of the invariant becomes strictly weaker, and so it still holds. Consider the $assignVal$ action preconditions. If the first clause in the new test, $(\forall b' < b)[b' \in dead]$, is satisfied, then the invariant obviously holds. So suppose that the second clause in the new test is satisfied:

$$(\exists b'' < b)[val(b'') = v \wedge (\forall b', b'' < b' < b)[b' \in dead]].$$

Then consider ballot b''. By the inductive hypothesis, the invariant holds for b'', so we can conclude that, in the pre-state of the assign-val step, if $b''' < b''$ then either $val(b''') = val(b'')$ or $b''' \in dead$. Combining this with the second clause, we see that $(\forall b' < b)[val(b') = v \vee b' \in dead]$, as needed. □

5 Untimed Distributed Safety Implementation

In this section we present a distributed implementation of the $Cons$ specification. This implementation is designed to achieve the safety guarantees described in Section 3. It makes no guarantees about liveness or timing; these will be considered later in the paper, starting from Section **??**. The implementation described in this system is completely timing-independent; timing constraints are not needed for the safety properties.

This part of the algorithm corresponds to the core Paxos algorithm;

5.1 Data Types

We constrain a previously-defined data type by adding some new structure, and define a new message data type:

- BId, the *ballot identifiers*; each of these is a record with fields $seqno \in \mathbb{N}$ and $procid \in I$.
- M, the set of *messages*; each of these is a record with fields $proposed \subseteq V$, $ballots \subseteq BId$, $val \in Bid \to V_\bot$, $voted \in I \to 2^{BId}$, and $abstained \in I \to 2^{BId}$.
- $m_0 \in M$, a distinguished *empty message*, with $m_0.proposed = \emptyset$, $m_0.ballots = \emptyset$, $m_0.val(b) = \bot$ for all b, $m_0.voted(i) = \emptyset$ for all i, and $m_0.abstained = \emptyset$ for all i.
- If $m, m' \in M$, we define $m \leq m'$ provided that $m.proposed \subseteq m'.proposed$, $m.ballots \subseteq m'.ballots$, $m.val(b) \in \{m'.val(b), \bot\}$ for all b, $m.voted(i) \subseteq m'.voted(i)$ for all i, and $m.abstained(i) \subseteq m'.abstained(i)$ for all i.
- If $m, m' \in M$, we define $m+m'$ to be the message $m'' \in M$ such that $m''.proposed = m.proposed \cup m'.proposed$, $m''.ballots = m.ballots \cup m'.ballots$, $m''.val(b) = m.val(b) if m.val(b) \neq \bot$, else $m'.val(b)$, for all b, $m''.voted(i) = m.voted(i) \cup m'.voted(i)$ for all i, and $m''.abstained(i) = m.abstained(i) \cup m'.abstained(i)$ for all i.

5.2 Overview

The system consists of a composition of an I/O automaton, $Paxos(i)$ for each $i \in I$, plus a set of lossy point-to-point channels, $Channel(i, j)$, between the various processes in I, plus a global $BallotTrigger$ service that tells all the $Paxos(i)$ automata when they should start new ballots.

A $\mathsf{fail}(i)$ event may occur at endpoint i. When this happens, $Paxos(i)$ stops operating. (That is, it enters a state from which no further locally-controlled actions are enabled and from which inputs have no effects.) In addition, $\mathsf{fail}(i)$ is provided as an input to $BallotTrigger$, which will also exhibit failure behavior at endpoint i.

$Channel(i, j)$ handles communication from endpoint i to endpoint j. A $\mathsf{lose}(i, j)$ event may occur on $Channel(i, j)$. This has the effect of losing some of the messages in transit. Channels do not generate new messages spontaneously. A send input event places the indicated message in the channel, and the channel delivers messages with a recv event.

In the rest of this section, we describe the various components in more detail.

5.3 Channels

Figure 4 provides the code for $Channel(i, j)$ (possibly $i = j$). Since we are dealing here with safety properties only, we do not make any assumptions about message delivery time.

Signature:

Inputs:
  $\mathsf{send}(m, i, j)$, $m \in M$
  $\mathsf{lose}(i, j)$

Outputs:
  $\mathsf{recv}(m, i, j)$, $m \in M$

State:

  $msgs$, a multiset of elements of M, initially empty

Transitions:

Input $\mathsf{send}(m, i, j)$
Effect:
  add one copy of m to $msgs$

Output $\mathsf{recv}(m, i, j)$
Precondition:
  $m \in msgs$
Effect:
  remove one copy of m from $msgs$

Input $\mathsf{lose}(i, j)$
Effect:
  remove any sub-multiset from $msgs$

Figure 4: $Channel(i, j)$ automaton

5.4 BallotTrigger

The $BallotTrigger$ service encapsulates all the computation needed to determine when a $Paxos(i)$ automaton should start a new ballot. This computation includes a leader election algorithm, designed to determine *who* should start a ballot, plus timeouts designed to determine *when* the leader should start a new ballot (because previous attempts appear to have failed). For the purpose of this section, we ignore the inner workings of the $BallotTrigger$ service, because they are not needed to prove the safety property. Instead, we assume simply that the $BallotTrigger$ service has a $\mathsf{new\text{-}ballot}(i)$ output action for each i, which it uses to tell $Paxos(i)$ to start a new ballot. The inner workings of the $BallotTrigger$ service are described in detail in Section 6.3.

5.5 Paxos(i)

$Paxos(i)$ is the piece of the distributed Paxos algorithm that runs at node i. Given that the $BallotTrigger$ service says when to start new ballots, then all the $Paxos(i)$ processes have to do is:

1. Start a new ballot when told to do so by $BallotTrigger(i)$.
2. Propagate the needed ballot information (in the background).
3. Abstain from ballots, when this is allowed.
4. Vote for ballots, when allowed.
5. Decide when they have enough information to (a) assign a value to a ballot and (b) determine that a decision has been made. These two, (a) and (b), constitute the two phases of a "ballot operation".

The needed information is propagated in the background. This information includes values that have been proposed, ballots that have been started, values that have been assigned to ballots, and who has voted for and abstained from ballots.

Figure 5 contains the signature and the state for $Paxos(i)$.

The definition of $dead$ used here is a local version of the notion of "dead" used in the $Global1$ and $Global2$ algorithms above. That is defined by using the "real" abstention information, not just the information known locally.

Figure 6 shows the transitions of the $Paxos(i)$ automaton. An init transition simply records the submitted value and changes the mode to active. The new-ballot input action tells $Paxos(i)$ that it should start up a new ballot; $Paxos(i)$ simply records this request.

Once a request for a new ballot has arrived, the new ballot is started by $Paxos(i)$ with a make-ballot action. This action selects a new sequence number that is bigger than any sequence number it has previously learned about. The new ballot identifier b is a combination of this sequence number and the process identifier i. The effect of this transition is to designate the new ballot identifier b as "known". No value is associated with b at this point.

$Paxos(i)$ uses an abstain action to abstain from all the ballots in some set B. It is allowed to do this provided that it knows about some ballot with an identifier larger than any in B, and provided that it hasn't already voted for any of the ballots in B.

After a ballot has been started, a value for the ballot has to be chosen. $Paxos(i)$ uses an assign-val transition to assign a value v to a ballot b. The ability to assign v to ballot b depends on an important consistency check with smaller ballots. Specifically, $Paxos(i)$ checks that b is a known ballot, and that i is the process that originally started b. Also, no value has yet been assigned to b (as far as i knows, but since i is the owner of b, this will mean that no value has in fact been assigned to b anywhere). The value v must be known to be someone's initial value. Moreover, all smaller ballots either have the same value v, or are known to be dead.

In order to decide on a consensus value given by the value assigned to a ballot, processes of the system have to vote for that ballot. $Paxos(i)$ may vote for a ballot b if b is a ballot that is known to have a certain value, and if i has not already abstained from b. This vote is cast by means of the vote(b, i) action.

Once enough votes are cast for a ballot b whose value is v, $Paxos(i)$ may decide (internally) on value v for consensus, with an internal-decide(b, v) transition. Enough votes means a write-quorum has voted for b. (Note that any process may decide in this way—not just the originator of the ballot.) Finally, $Paxos(i)$ may announce the decision to the external environment with a decide(i) action anytime after it has learned the decision.

Signature:

Inputs:
$\mathsf{init}(v, i)$, $v \in V$
$\mathsf{new\text{-}ballot}(i)$
$\mathsf{recv}(m, j, i)$, $m \in M$, $j \in I$
$\mathsf{fail}(i)$

Outputs:
$\mathsf{decide}(v, i)$, $v \in V$
$\mathsf{send}(m, i, j)$, $m \in M$, $j \in I$

Internal:
$\mathsf{make\text{-}ballot}(b, i)$, $b \in BId$
$\mathsf{abstain}(B, i)$, $B \subseteq BId$, $B \neq \emptyset$
$\mathsf{assign\text{-}val}(b, v, i)$, $b \in BId$, $v \in V$
$\mathsf{vote}(b, i)$, $b \in BId$
$\mathsf{internal\text{-}decide}(b, v, i)$, $b \in BId$, $v \in V$

State:

$mode \in \{idle, active\}$, initially $idle$

$proposed \subseteq V$, initially $\emptyset$
$ballots \subseteq BId$, initially $\emptyset$
$val \in BId \to V_\perp$, initially everywhere $\perp$
$voted \in I \to 2^{BId}$, initially everywhere $\emptyset$
$abstained \in I \to 2^{BId}$, initially everywhere $\emptyset$
$failed$, a Boolean, initially $false$

$do\text{-}make\text{-}ballot$, a Boolean, initially $false$
$succeeded \subseteq BId$, initially $\emptyset$
$done$, a Boolean, initially $false$

Derived variables:

$state\text{-}message \in M$, defined as m where
$m.proposed = proposed$,
$m.ballots = ballots$,
$m.val = val$,
$m.voted = voted$, and
$m.abstained = abstained$.

$dead \subseteq BId$, defined as $\{b \in BId : (\exists R \in read\text{-}quorums)(\forall j \in R)[b \in abstained(j)]\}$.

Figure 5: $Paxos(i)$: Signature and state variables

The send and recv actions are used for gossiping information among the processes in the configuration performing consensus. The information being sent around includes the $proposed$ and $ballots$ sets, and the $value$, $voted$ and $abstained$ maps.

To prove safety, we map $Paxos$ to $Global2$. The mapping is:

- $\forall i \in I(i \in Global2.initiated \Leftrightarrow Paxos(i).mode = active)$.
- $\forall i \in I(i \in Global2.failed \Leftrightarrow Paxos(i).failed)$.
- $Global2.proposed = \cup_{i \in I} Paxos(i).proposed$.
- $Global2.ballots = \cup_{i \in I} Paxos(i).ballots$.
- $Global2.succeeded = \cup_{i \in I} Paxos(i).succeeded$.
- $\forall i \in I(Global2.decided \Leftrightarrow Paxos(i).done)$.
- $\forall b \in BId(Global2.val(b) = Paxos(b.procid).val(b))$.
- $\forall i \in I(Global2.voted(i) = Paxos(i).voted(i))$.
- $\forall i \in I(Global2.abstained(i) = Paxos(i).abstained(i))$.

It was fairly trivial to prove this mapping holds: every action done by a Paxos automaton corresponds to the same-named action in $Global2$, except for $doMakeBallot$, which corresponds to the empty execution. There are some actions in $Global2$, such as $internalDecide$ that are not associated with a particular node. For these, whenever $internalDecide$ was fired in Paxos on any node, we fired the global $internalDecide$ in $Global2$. This is possible because $Global2$ allows for repeats of previously performed internal actions, so multiple $internalDecide$ on the same ballot is acceptable.

To prove that this mapping works, we needed some invariants:

1. $\forall i, j \in I,\ Paxos(j).abstained(i) \subseteq Paxos(i).abstained(i)$.
   The information some node has of another node's abstained set is true, although not necessarily complete.
2. $\forall i, j \in I\ Paxos(j).voted(i) \subseteq Paxos(i).voted(i)$.
   The information some node has of another node's voted set is true, although not necessarily complete.
3. $\forall i \in I, b \in BId,\ Paxos(i).val(b) \neq \perp \Rightarrow Paxos(i).val(b) = Paxos(b.procid).val(b)$.
   The information some node has of the value of a ballot is consistent with the value assigned by the node that made the ballot.
4. $\forall i \in I, b \in BId,\ b \in Paxos(i).ballots \Rightarrow b \in Paxos(b.procid).ballots$.
   If a node knows of a ballot's existence, then the node that created the ballot also knows.
5. $\forall b \in BId,\ b \notin Paxos(b.procid).ballots \Rightarrow Paxos(b.procid).val(b) = \perp$.
   If a ballot has not been created, then its creator does not believe it exists.
6. $\forall m \in M,\ m \in Channel(i, j).msgs \Rightarrow m \leq Paxos(i).state\text{-}message$.
   Every message in a channel has the $\leq$ relationship with its sender that holds throughout the message's existence:

Output $\mathsf{send}(m, i, j)$
Precondition:
$\neg failed$
$mode = active$
$m \le state\text{-}message$
Effect:
none

Input $\mathsf{recv}(m, j, i)$
Effect:
if $\neg failed$ then
if $mode = active$ then
$proposed \leftarrow proposed \cup m.proposed$
$ballots \leftarrow ballots \cup m.ballots$
for $b \in BId$ do
if $m.val(b) \ne \perp$ then $val(b) \leftarrow m.val(b)$
for $k \in I$ do
$voted(k) \leftarrow voted(k) \cup m.voted(k)$
$abstained(k) \leftarrow abstained(k) \cup m.abstained(k)$

Input $\mathsf{init}(v, i)$
Effect:
if $\neg failed$ then
if $mode = idle$ then
$mode \leftarrow active$
$proposed \leftarrow proposed \cup \{v\}$

Input $\mathsf{new\text{-}ballot}(i)$
Effect:
if $\neg failed$ then
if $mode = active$ then
$do\text{-}make\text{-}ballot \leftarrow true$

Output $\mathsf{make\text{-}ballot}(b, i)$
Precondition:
$\neg failed$
$mode = active$
$do\text{-}make\text{-}ballot = true$
$(\forall b' \in ballots)[b.seqno > b'.seqno]$
$b.procid = i$
Effect:
$ballots \leftarrow ballots \cup \{b\}$
$do\text{-}make\text{-}ballot \leftarrow false$

Internal $\mathsf{abstain}(B, i)$
Precondition:
$\neg failed$
$mode = active$
$(\forall b \in B)(\exists b' \in ballots)[b < b']$
$(voted(i) \cup abstained(i)) \cap B = \emptyset$
Effect:
$abstained(i) \leftarrow abstained(i) \cup B$

Output $\mathsf{assign\text{-}val}(b, v, i)$
Precondition:
$\neg failed$
$mode = active$
$b \in ballots$
$b.procid = i$
$val(b) = \perp$
$v \in proposed$
$(\forall b' < b)[b' \in dead]$
$\vee (\exists b'' < b)[val(b'') = v \wedge (\forall b', b'' < b' < b)[b' \in dead]]$
Effect:
$val(b) \leftarrow v$

Internal $\mathsf{vote}(b, i)$
Precondition:
$\neg failed$
$mode = active$
$b \in ballots$
$val(b) \ne \perp$
$b \notin (voted(i) \cup abstained(i))$
Effect:
$voted(i) \leftarrow voted(i) \cup \{b\}$

Internal $\mathsf{internal\text{-}decide}(b, i)$
Precondition:
$\neg failed$
$mode = active$
$b \in ballots$
$(\exists W \in write\text{-}quorums)(\forall j \in W)[b \in voted(j)]$
Effect:
$succeeded \leftarrow succeeded \cup \{b\}$

Output $\mathsf{decide}(v, i)$
Precondition:
$\neg failed$
$mode = active$
$b \in succeeded$
$v = val(b)$
$\neg done$
Effect:
$done \leftarrow true$

Input $\mathsf{fail}(i)$
Effect:
$failed \leftarrow true$

Figure 6: $Paxos(i)$: Transitions

6 Timed Distributed Implementation

In this section, we constrain the implementation described earlier in the paper with various timing conditions. In the next section, we prove latency and liveness guarantees for the constrained algorithm, in situations where failure and timing behavior stabilizes from some point onward.

The timing constraints we describe are of several kinds:

1. We define a specific timing-based implementation of the $BallotTrigger$, which tells the $Paxos(i)$ automata when to start ballots. The $BallotTrigger$ implementation performs failure detection using timeouts, in order to elect a leader, and monitors progress in the main $Paxos$ algorithm in order to determine when the leader should start a new ballot.

2. We add time bounds for message delivery to the $Channel(i, j)$ automata.

3. We restrict message sending so that it occurs only in two situations: we send information when it is first generated, and we allow periodic gossip, according to a specific time interval.

4. We assume that each $Paxos(i)$ performs its enabled locally-controlled actions without any time-passage.

In order to express these constraints, we give explicit timing-dependent code for the $BallotTrigger$. We rewrite the channel code to include the new timing constraint. We modify the code for $Paxos_i$ to include the scheduling of message sends and of other locally-controlled actions. Thus, we give a complete description of all the timing assumptions of the system in terms of automata.

We define the entire system as the composition of the free versions of all the components. In our latency analysis, however, we consider situations where, from some point onward, the execution is regular, and other "good behavior" occurs (for example, no fail or lose events).

6.1 Data Types and Constants

We assume:

- $d \in R^{>0}$, the message delay.

6.2 Timed Channel

The revised model for the channel, incorporating a message delay of d, appears in Figure 7. Note that this channel model still tolerates losses.

6.3 The Ballot Trigger

The $BallotTrigger$ service times out processes when they appear to have failed. It can issue a $\mathsf{new\text{-}ballot}(i)$ event as soon as it determines that i is the leader (the process with the highest id that does not yet appear to have failed). Then it can time out ballots that it has tried to start: it measures time, starting with the $\mathsf{new\text{-}ballot}(i)$ event, seeing if $\mathsf{assign\text{-}val}(*, *, i)$ and $\mathsf{decide}(*, i)$ events occur within the amount of time they should. If not, and if i is still deemed to be the leader, then the $BallotTrigger$ service issues another $\mathsf{new\text{-}ballot}(i)$ event.

The $BallotTrigger$ service receives as inputs the init, assign-val, and decide actions of the $Paxos$ automata. In this way, $BallotTrigger$ can monitor what $Paxos$ is doing. The service implements a failure detector, and uses the results of failure detection to decide who is the current leader. The current leader triggers periodic $new\text{-}ballot$ events.

Signature:

Inputs:
  $\mathsf{send}(m, i, j)$, $m \in M$
  $\mathsf{lose}(i, j)$

Outputs:
  $\mathsf{recv}(m, i, j)$, $m \in M$

Time-passage:
  $\nu(t)$, $t \in R^{\geq 0}$

State:

$msgs$, a multiset of elements of $M \times R$, initially empty
$clock \in R^{\geq 0}$, initially 0

Transitions:

Input $\mathsf{send}(m, i, j)$
Effect:
  add one copy of $(m, clock + d)$ to $msgs$

Output $\mathsf{recv}(m, i, j)$
Precondition:
  $m \in msgs$
Effect:
  remove one copy of $(m, *)$ from $msgs$

Input $\mathsf{lose}(i, j)$
Effect:
  remove any sub-multiset from $msgs$

Time-passage $\nu(t)$
Precondition:
  $(\forall (m, t') \in msgs)[clock + t \leq t']$
Effect:
  $clock \leftarrow clock + t$

Figure 7: $Channel(i, j)$ automaton

$BallotTrigger$ consists of a collection of timed I/O automata, $BT(i)$, $i \in I$, together with a communication service. The code of $BT(i)$ is given in Figure 8.

To justify the $2d + \epsilon$ bound in the init code: This is based on assuming that init actions happen at most d apart, which is something that we depend on the Recon service to guarantee, in the Rambo paper. The second d comes from delivery in this algorithm, and ϵ is to avoid a race between arrival from a live process and timing it out.

6.4 Timed Paxos(i)

Our version of $Paxos(i)$ is very nondeterministic, in particular, in what messages it sends and when. However, only a subset of the enabled sends are actually important for the progress of the algorithm. Therefore, for the purpose of latency analysis, we restrict communication so only certain "important" sends occur, in addition to periodic gossip.

Now we give the modifications to the main $Paxos(i)$ algorithm. These modifications fix the message sends to occur only at certain times—when new information is generated and should be communicated to everyone, and periodically, at intervals of d. Also, they specify that non-send locally controlled actions that are enabled occur without any time-passage.

6.5 The Complete Timed System

The complete system we consider consists of timed automata for all the nodes, channels, and the ballot trigger, specifically:

1. The timed automata $\mathit{free}(Paxos(i))$ for all $i \in I$.
2. The timed automata $\mathit{free}(BT(i))$ for all $i \in I$.
3. The timed automata $\mathit{free}(Channel(i, j))$ for all $i, j \in I$.

Signature:

Input:
- $\mathsf{init}(v, i), v \in V$
- $\mathsf{assign\text{-}val}(b, v, i), b \in BId, v \in V$
- $\mathsf{decide}(v, i), v \in V$
- $\mathsf{recv}(\mathsf{alive}, j, i), j \in I, j \neq i$
- $\mathsf{fail}(i)$

Output:
- $\mathsf{new\text{-}ballot}(i)$
- $\mathsf{send}(\mathsf{alive}, i, j), j \in I, j \neq i$

Internal:
- $\mathsf{node\text{-}timeout}(j, i), j \in I, j \neq i$

Time-passage:
- $\nu(t), t \in R^{>0}$

State:

$mode \in \{idle, active\}$, initially $idle$
$suspected$, a finite subset of I, initially $\emptyset$
$done$, a Boolean, initially $false$
$clock \in R^{\geq 0}$, initially 0
$failed$, a Boolean, initially $false$
$timeout \in I \to (R^{>0} \cup \{\infty\})$, initially everywhere ∞
$next\text{-}ballot\text{-}time \in R^{>0} \cup \{\infty\}$, initially ∞
$next\text{-}send\text{-}time \in I \to (R^{>0} \cup \{\infty\})$, initially everywhere 0

Derived variables:

$leader \in I_\perp$, defined as $\perp$ if $suspected = I$, else $\max(I - suspected)$.

Transitions:

Output $\mathsf{send}(\mathsf{alive}, i, j)$
Precondition:
  $\neg failed$
  $mode = active$
  $j \notin suspected$
Effect:
  $next\text{-}send\text{-}time(j) \leftarrow clock + d$

Input $\mathsf{recv}(\mathsf{alive}, j, i)$
Effect:
  if $\neg failed$ then
    if $mode = active$ then
      $timeout(j) \leftarrow clock + 2d + \epsilon$
      if $j \in suspected$ then
        $next\text{-}send\text{-}time(j) \leftarrow clock$
      $suspected \leftarrow suspected - \{j\}$
      if $i \neq leader$ then $next\text{-}ballot\text{-}time \leftarrow \infty$

Input $\mathsf{init}(v, i)$
Effect:
  if $\neg failed$ then
    if $mode = idle$ then
      $mode \leftarrow active$
      for every $j \in I - \{i\}$ do
        $next\text{-}send\text{-}time(j) \leftarrow clock$
        $timeout(j) \leftarrow clock + 2d + \epsilon$
      if $i = leader$ then $next\text{-}ballot\text{-}time \leftarrow clock$

Internal $\mathsf{node\text{-}timeout}(j, i)$
Precondition:
  $\neg failed$
  $mode = active$
  $clock = timeout(j)$
Effect:
  $suspected \leftarrow suspected \cup \{j\}$
  $timeout(j) \leftarrow \infty$
  if $i = leader$ and $i < j$ and $\neg done$ then
    $next\text{-}ballot\text{-}time \leftarrow clock$

Input $\mathsf{assign\text{-}val}(b, v, i)$
Effect:
  if $\neg failed$ then
    if $mode = active$ then
      $next\text{-}ballot\text{-}time \leftarrow clock + 2d + \epsilon$

Input $\mathsf{decide}(v, i)$
Effect:
  if $\neg failed$ then
    if $mode = active$ then
      $done \leftarrow true$
      $next\text{-}ballot\text{-}time \leftarrow \infty$

Output $\mathsf{new\text{-}ballot}(i)$
Precondition:
  $\neg failed$
  $mode = active$
  $clock = next\text{-}ballot\text{-}time$
  $\neg done$
Effect:
  $next\text{-}ballot\text{-}time \leftarrow clock + 2d + \epsilon$

Time-passage $\nu(t)$
Precondition:
  if $\neg failed$ then
    $clock + t \leq next\text{-}ballot\text{-}time$
    $\forall j \in I$
      $clock + t \leq timeout(j)$ and
      $clock + t \leq next\text{-}send\text{-}time(j)$
Effect:
  $clock \leftarrow clock + t$

Input $\mathsf{fail}(i)$
Effect:
  $mode \leftarrow failed$

Figure 8: $BT(i)$: Automaton

Additions to signature:

Internal:
$\mathsf{gossip}(i)$

Time-passage:
$\nu(t), t \in R^{>0}$

Additions to state:

for every $j \in I - \{i\}$:
$sched\text{-}msg(j) \in M$, initially m_0

$clock \in R^{\geq 0}$, initially 0
$next\text{-}gossip\text{-}time \in R^{\geq 0}$, initially 0

Changes to transitions:

Output $\mathsf{send}(m, i, j)$
Precondition:
  $\neg failed$
  $mode = active$
  $m = sched\text{-}msg(j)$
Effect:
  $sched\text{-}msg(j) \leftarrow m_0$

Input $\mathsf{init}(v, i)$
Effect:
  if $\neg failed$ then
   if $mode = idle$ then
    as before, plus:
    for $j \in I - \{i\}$ do
     $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + m$ where
      $m = m_0$ except that $m.proposed = \{v\}$

Output $\mathsf{make\text{-}ballot}(b, i)$
Precondition:
  as before
Effect:
  as before, plus:
  for $j \in I - \{i\}$ do
   $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + m$ where
    $m = m_0$ except that $m.ballots = \{b\}$

Internal $\mathsf{abstain}(B, i)$
Precondition:
  as before
Effect:
  as before, plus:
  for $j \in I - \{i\}$ do
   $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + m$ where
    $m = m_0$ except that $m.abstained(i) = B$

Output $\mathsf{assign\text{-}val}(b, v, i)$
Precondition:
  as before
Effect:
  as before, plus:
  for $j \in I - \{i\}$ do
   $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + m$ where
    $m = m_0$ except that $m.val(b) = v$

Internal $\mathsf{vote}(b, i)$
Precondition:
  as before
Effect:
  as before, plus:
  for $j \in I - \{i\}$ do
   $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + m$ where
    $m = m_0$ except that $m.voted(i) = \{b\}$

Time-passage $\nu(t)$
Precondition:
  if $\neg failed$ then
   No make-ballot, abstain, assign-val,
    vote, internal-decide, or decide action is enabled.
   $\forall j \in I - \{i\}, sched\text{-}msg(j) = m_0$
   $clock + t \leq next\text{-}gossip\text{-}time$
Effect:
  $clock \leftarrow clock + t$

$\mathsf{gossip}(i)$
Precondition:
  $\neg failed$
  $clock = next\text{-}gossip\text{-}time$
Effect:
  for every $j \in I - \{i\}$ do
   $sched\text{-}msg(j) \leftarrow sched\text{-}msg(j) + state\text{-}msg$
  $next\text{-}gossip\text{-}time \leftarrow next\text{-}gossip\text{-}time + d$

Figure 9: Timed $Paxos(i)$: Modifications to $Paxos(i)$

Because we use the free versions of the automata, we are not insisting that the timing requirements actually be satisfied. However, our analysis in the following section will assume executions in which they are satisfied from some point onward.

7 Latency Bounds and Liveness

7.1 Stability Assumptions

We consider an execution α of the complete timed system, and a finite prefix α' of α. We assume that α is *stable* after α', specifically:

1. For all i, $\alpha|Paxos(i)$ is regular after $\alpha'|Paxos(i)$.
   This implies that all the send events happen when they should, and that local processing time is 0.

2. For all i, $\alpha|BT(i)$ is regular after $\alpha'|BT(i)$.
   This implies that all the timeouts for failure detection and for starting a new ballot happen when they should.

3. For all $i, j \in I$, $i \neq j$, $\alpha|Channel(i,j)$ is regular after $\alpha'|Channel(i,j)$.
   This implies that messages that don't get lost get delivered within time d.

4. No fail events happen in α after α'.

5. No lose events happen in α after α'.

7.2 Latency Bound

In addition, we need some special assumptions particular to Paxos, to prove a latency bound:

Theorem 7.1 *Consider an admissible*[1] *timed execution α of the system and a prefix α' of α. Suppose that:*

1. *α is stable after α', as defined by the properties listed just above.*

2. *For every i, if $fail(i)$ does not occur in α, then some* $\mathsf{init}(*, i)$ *occurs in α.*
   (That is, every process that doesn't fail initiates the consensus algorithm.)

3. *No* init *events occur in α after α'.*

4. *There exist $R \in read\text{-}quorums$ and $W \in write\text{-}quorums$ such that for all $i \in R \cup W$, no* fail_i *occurs in α.*

Then for every i such that $\mathsf{fail}(i)$ *does not occur, a* $\mathsf{decide}(*, i)$ *event occurs by time $\ell time(\alpha') + 10d + \epsilon$.*

Proof. (Sketch:) Let S be the set of processes that don't fail in α. Let ℓ be $\max(S)$.

By time $\ell time(\alpha') + 2d$, all BT processes that don't fail have correct information about failures, that is, have $suspected = I - S$, and keep this correct information forever thereafter in α. So, after this time, only the "final leader" ℓ ever performs a make-ballot.

Then within an additional time $2d$, that is, by time $\ell time(\alpha') + 4d$, $Paxos(\ell)$ has learned about all ballots that anyone in S is ever going to learn about, from among those started by processes other than ℓ. So, after time $\ell time(\alpha') + 4d$, any ballot that is ever started (by ℓ) is guaranteed to get a *seqno* that is strictly

[1] This means that the limit time is ∞

larger than the $seqno$ of any ballot that anyone in S is ever going to learn about, from among those started by processes other than ℓ. That is because the choice of sequence number for such a new ballot will take all these earlier sequence numbers into account.

Then within $2d + \epsilon$, the leader will start a new ballot, if a previously one doesn't succeed first. And then within an additional $4d$, this will complete. $\square$

References

[1] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Earlier version in Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[3] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[4] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on DIStributed Computing (DISC)*, Toulouse, France, October 2002. To appear. Also, Technical Report MIT-LCS-TR-856.

[5] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. Technical Report MIT-LCS-TR-856, MIT Laboratory for Computer Science, Cambridge, MA, 2002. Also, in [4].

[6] Leslie Lamport. The part-time parliament. Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[7] Butler Lampson et al. POCS Course Notes (Principles of Computer Systems), 1999. Available online at `research.microsoft.com/lampson/48-POCScourse/Abstract.html`.

[8] Roberto DePrisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms* 11th International Workshop, WDAG'97, Saarbrücken, Germany, September 1997 Proceedings, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125, Berlin-Heidelberg, 1997. Springer-Verlag.

[9] Roberto DePrisco. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1997. Also, MIT/LCS/TR-717.

[10] Roberto DePrisco, Butler Lampson, and Nancy Lynch. Fundamental study: Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243:35–91, 2000. Earlier version appeared in [8].

[11] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. Submitted for publication. Also, Technical Report, Distributed Programming Laboratory, Swiss Federal Institute of Technology, Lausanne, January 2001.

[12] Leslie Lamport and Eli Gafni. Disk Paxos. *Distributed Computing*. To appear. Also appeared as SRC Research Report 163 (July, 2000), and *Distributed Computing: 14th International Conference, DISC 2000*, Maurice Herlihy, editor. Lecture Notes in Computer Science number 1914, Springer-Verlag, (2000) 330-344.

[13] G. Chockler and D. Malkhi. Active disk Paxos with infinitely many processes. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, Monterey, CA, July 2002.

[14] Butler Lampson. The ABCD's of paxos. In *Proceedings of the Twentieth Annual ACM symposium on Principles of Distributed Computing*, Newport, RI, August 2001.

[15] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001.

[16] Nancy Lynch. Implementing atomic objects in a dynamic environment. In *Proceedings of the Twentieth ACM Annual Symposium on Distributed Computing (Leslie Lamport's 60th Birthday Celebration)*, Newport, RI, August 2001.

[17] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[18] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.