Fast Paxos

Leslie Lamport

14 July 2005
Revised 18 January 2006
Minor revision 14 April 2006

MSR-TR-2005-112

To appear in *Distributed Computing.*

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

Abstract

As used in practice, traditional consensus algorithms require three message delays before any process can learn the chosen value. Fast Paxos is an extension of the classic Paxos algorithm that allows the value to be learned in two message delays. How and why the algorithm works are explained informally, and a TLA$^+$ specification of the algorithm appears as an appendix.

Contents

1 Introduction

The consensus problem requires a set of processes to choose a single value. This paper considers the consensus problem in an asynchronous message-passing system subject to non-Byzantine faults. A solution to this problem must never allow two different values to be chosen despite any number of failures, and it must eventually choose a value if enough processes are non-faulty and can communicate with one another.

In the traditional statement of the consensus problem, each process proposes a value and the chosen value must be one of those proposed values. It is not hard to see that any solution requires at least two message delays before any process learns what value has been chosen [3]. A number of algorithms achieve this delay in the best case. The classic Paxos algorithm [7, 9] is popular because it achieves the optimal delay in the normal case when used in practical systems [12].

The apparently optimal number of message delays required by traditional consensus algorithms is illusory—an artifact of the traditional problem statement in which values are chosen by the same processes that propose them. In many applications, values are not proposed by the same processes that choose the value. For example, in a client/server system, the clients propose the next command to be executed and the servers choose one proposed command. When a traditional consensus algorithm is used in such a system, three message delays are required between when a client proposes a command and when some process learns which command has been chosen.

A fast consensus algorithm is one in which a process can learn the chosen value within two message delays of when it is proposed, even if values are proposed and chosen by different sets of processes. It has been shown that no general consensus algorithm can guarantee learning within two message delays if competing proposals collide—that is, if two different values are proposed concurrently [11]. A fast consensus algorithm therefore cannot always be fast in the event of collision.

Fast Paxos is a fast consensus algorithm that is a variant of classic Paxos. In the normal case, learning occurs in two message delays when there is no collision and can be guaranteed to occur in three message delays even with a collision. Moreover, it can achieve any desired degree of fault tolerance using the smallest possible number of processes.

The basic idea behind Fast Paxos also underlies an earlier algorithm of Brasileiro et al. [1]. However, they considered only the traditional consensus problem, so they failed to realize that their algorithm could be easily modified to obtain fast consensus. Pedone and Schiper's R-Consensus Algorithm [15] can also be modified to yield a fast consensus algorithm. However, the resulting algorithm requires at least four message delays in the presence of collision. (They also solve a more general problem with a similar algorithm that, in the special case of consensus, also requires at least four message delays when a collision occurs [14].) Zielinski [16] recently presented a fast consensus algorithm that can be viewd as a variant of Fast Paxos.

Fast Paxos is at heart a simple extension of classic Paxos. It is easy to understand why Fast Paxos works if one understands why classic Paxos works. I therefore begin in Section 2 by explaining classic Paxos. Section 3 then explains how to modify classic Paxos to obtain Fast Paxos. The exposition in these sections is somewhat informal. Their goal is to explain why the algorithms work; they do not provide the customary concise descriptions of the algorithms in pseudocode or any other language. Precise statements of the classic Paxos algorithm can be found elsewhere [4, 7], and a formal TLA$^+$ [10] specification of Fast Paxos is given in the appendix. The TLA$^+$ specification is general enough to encompass all the variants discussed here. However, no single statement of Fast Paxos adequately describes the details involved in implementing all these variants. A thorough understanding of the principles underlying Fast Paxos is better preparation for implementing it than any description of the algorithm, be it in pseudocode or a formal language.

A concluding section discusses the optimality of Fast Paxos, explains the relation between the algorithm of Brasileiro et al. and Fast Paxos, and briefly mentions the generalization of classic and Fast Paxos to handle Byzantine failures.

2 The Classic Paxos Algorithm

2.1 The Problem

The consensus problem is most usefully expressed in terms of three sets of agents: *proposers* that can propose values, *acceptors* that choose a single value, and *learners* that learn what value has been chosen. An agent represents a role played by some process; a single process can play multiple roles. For example, in a client/server system, a client might play the roles of proposer and learner, and a server might play the roles of acceptor and learner.

I assume the customary asynchronous, distributed, non-Byzantine model of computation in which:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. However, an agent may not perform an incorrect action.
- Agents communicate by sending messages that can take arbitrarily long to be delivered, can be delivered out of order, can be duplicated, and can be lost. However, messages are never (undetectably) corrupted.

The safety requirements for consensus are as follows, where a value can be proposed only by a proposer.

Nontriviality	Only proposed values can be learned.
Consistency	At most one value can be learned.

Most consensus algorithms, including Fast Paxos, trivially satisfy the nontriviality requirement. I will therefore largely ignore nontriviality. Consistency is satisfied by ensuring that only a single value is chosen by the acceptors, and that only a value they choose can be learned. Ensuring that only one value can be chosen is the hard part; it will be obvious how a chosen value should be learned. So, I will concentrate on how the acceptors choose a value.

The safety requirements must be maintained in the face of any number of (non-Byzantine) failures. A consensus algorithm should also satisfy a progress requirement stating approximately that a value is eventually chosen if enough agents are nonfaulty. We do not want to rely on proposers or learners for progress, because they could be unreliable. For example, we don't want a client/server system to halt just because a client fails to respond; and a client might play the role of proposer and/or learner. Progress should require only that enough acceptors are nonfaulty, as long as there is a nonfaulty proposer to propose a value and a nonfaulty learner to learn one. However, such a requirement is problematic because the classic Fischer, Lynch, Paterson result [5] implies that it cannot be satisfied by any fault-tolerant asynchronous algorithm that satisfies the safety properties of consensus. Therefore, some additional hypothesis is required. Different consensus algorithms use different hypotheses. I will defer a precise statement of the Paxos algorithm's progress property, which involves defining what "nonfaulty" means, until I have described how it satisfies the safety properties. This is possible because the safety requirements must hold without any nonfaultiness assumptions.

Progress must be possible if all the agents are nonfaulty, even if they have all failed and been restarted. Since a value could have been learned before the agents failed, agents must have some stable storage that survives failure and restart. I assume that an agent restores its state from stable storage when it restarts, so the failure of an agent is indistinguishable from its simply pausing. There is thus no need to model failures explicitly.

2.2 Safety

2.2.1 The Basic Algorithm

I begin by describing a simple version of the Paxos consensus algorithm that satisfies the safety requirements of consensus. It is extended in Section 2.3 to the complete algorithm that also satisfies a progress property.

The algorithm executes multiple rounds, where each round is numbered by a positive integer. Rounds are not necessarily executed in numerical order, individual rounds need not be completed and may be skipped altogether, and different rounds may be executed concurrently. A round may choose a value. A value is defined to be chosen iff it is chosen in some round. In each round, an acceptor may vote to accept some single value or it may decide not to vote. A value v is defined to be chosen in a round iff a majority of the acceptors vote in that round to accept v. For simplicity, I describe an algorithm that never terminates but continues to execute rounds even after a value has been chosen. Termination and other optimizations are discussed in Section 2.4 below.

Achieving consistency requires that two different values not be chosen. Since an acceptor votes to accept at most one value in any round and any two majorities contain an acceptor in common, it is impossible for two different values to be chosen in the same round. However, an acceptor can vote to accept different values in different rounds. The hard part of achieving consistency is ensuring that different values are not chosen in different rounds.

Although we reason about the algorithm in terms of all the votes that an acceptor has ever cast, the acceptor does not have to remember those votes. An acceptor a maintains only the following data:

$rnd[a]$	The highest-numbered round in which a has participated, initially 0. (Since 0 is not a round number, $rnd[a] = 0$ means that a has not participated in any round.)
$vrnd[a]$	The highest-numbered round in which a has cast a vote, initially 0. (Hence, $vrnd[a] \leq rnd[a]$ is always true.)

$vval[a]$ The value a voted to accept in round $vrnd[a]$; its initial value (when $vrnd[a] = 0$) is irrelevant.

Paxos assumes a set of *coordinator* agents. The roles of coordinator and acceptor are usually played by the same processes. For each round i, some coordinator is pre-assigned to be the coordinator of round i. Moreover, each coordinator is assigned to be the coordinator for infinitely many rounds. Proposers send their proposals to the coordinators. As explained below, the coordinator for round i picks a value that it tries to get chosen in that round. Each coordinator c maintains the following data:

$crnd[c]$ The highest-numbered round that c has begun, initially 0.

$cval[c]$ The value that c has picked for round $crnd[c]$, or the special value *none* if c has not yet picked a value for that round. Its initial value is irrelevant.

Round i progresses in the following phases, where c is the round's coordinator.

1. (a) If $crnd[c] < i$, then c starts round i by setting $crnd[c]$ to i, setting $cval[c]$ to *none*, and sending a message to each acceptor a requesting that a participate in round i.

   (b) If an acceptor a receives a request to participate in round i and $i > rnd[a]$, then a sets $rnd[a]$ to i and sends coordinator c a message containing the round number i and the current values of $vrnd[a]$ and $vval[a]$.

   If $i \leq rnd[a]$ (so a has begun round i or a higher-numbered round), then a ignores the request.

2. (a) If $crnd[c] = i$ (so c has not begun a higher-numbered round), $cval[c] = none$ (so c has not yet performed phase 2a for this round), and c has received phase 1b messages for round i from a majority of the acceptors; then by a rule described below, c uses the contents of those messages to pick a value v, sets $cval[c]$ to v, and sends a message to the acceptors requesting that they vote in round i to accept v.

   (b) If an acceptor a receives a request to vote in round i to accept a value v, and $i \geq rnd[a]$ and $vrnd[a] \neq i$; then a votes in round i to accept v, sets $vrnd[a]$ and $rnd[a]$ to i, sets $vval[a]$ to v, and sends a message to all learners announcing its round i vote.

If $i < rnd[a]$ or $vrnd[a] = i$ (so a has begun a higher-numbered round or already voted in this round), then a ignores the request.

A learner learns a value v if, for some round i, it receives phase 2b messages from a majority of acceptors announcing that they have all voted for v in round i.

A coordinator can execute a phase 1a action for a new round number at any time. However, the enabling condition for that action prevents a coordinator from starting a round with a lower number than one it has already started. Different rounds can be executed concurrently, but an acceptor will stop participating in a round if it receives (and acts on) a message for a higher-numbered round. Phase 2a messages requesting acceptance of different values can be sent in different rounds. However, the enabling condition for the phase 2a action and the unique assignment of rounds to coordinators ensure that phase 2a messages with different values cannot be sent for the same round.

2.2.2 Picking a Value in Phase 2a

I have not yet described the heart of the algorithm—namely, how the coordinator picks the value v in phase 2a. How it does this is derived from the requirement that the algorithm maintain the following fundamental property:

CP. For any rounds i and j with $j < i$, if a value v has been chosen or might yet be chosen in round j, then no acceptor can vote for any value except v in round i.

Of course, an equivalent statement of CP is:

For any rounds i and j with $j < i$, if an acceptor has voted for v in round i, then no value other than v has been or might yet be chosen in round j.

A value can be chosen in round i only if some acceptor votes for it. Therefore, CP implies that if $j < i$, then no value other than v can ever be chosen in round i if v is chosen in round j. CP thus implies that two different values cannot be chosen in two different rounds. We have already seen that two different values cannot be chosen in the same round, so CP implies consistency.

Since CP implies the consistency requirement, we can guarantee that the algorithm satisfies consistency by ensuring that it maintains CP. The only value v that an acceptor can vote for in round i is the one picked by the coordinator in phase 2a. So, we just need to ensure that v satisfies the following property:

> $CP(v, i)$: For any round $j < i$, no value other than v has been or might yet be chosen in round j.

Let a *majority set* be a set consisting of a majority of the acceptors. A value v has been or might be chosen in round j iff there is some majority set Q such that every acceptor in Q has either voted in round j for v or might yet do so. An acceptor a never decreases $rnd[a]$, and it ignores requests to vote in round j if $j < rnd[a]$. Therefore, we have:

> **Observation 1.** A value v has been or might be chosen in round j only if there is a majority set Q such that each acceptor a in Q has $rnd[a] \leq j$ or has voted for v in round j.

Observation 1 is false if "only if" is replaced with "iff" because $rnd[a] = j$ could be true if acceptor a has voted for some value other than v in round j. We can replace "only if" with "iff" if we also replace $rnd[a] \leq j$ with

$$(rnd[a] \leq j) \land (vrnd[a] < j)$$

but we won't need this more complicated condition.

Since any two majority sets have an acceptor in common, Observation 1 easily implies the following two observations:

> **Observation 2.** If there is a majority set Q such that every acceptor a in Q has $rnd[a] > j$ and has not voted in round j, then no value has been or ever might be chosen in round j.

> **Observation 3.** If there is a majority set Q and a value v such that every acceptor a in Q has $rnd[a] > j$ and has either voted for v in round j or has not voted in round j, then no value other than v has been or ever might be chosen in round j.

The hypotheses of Observation 3 do not imply that v has been or could be chosen in round j. In fact, those hypotheses could be satisfied by two different values v, in which case Observation 3 implies that no value has been or ever might be chosen in round j.

Suppose the coordinator has received round i phase 1b messages from a majority Q of acceptors. Since an acceptor a sets $rnd[a]$ to i upon sending a round i phase 1b message and it never decreases $rnd[a]$, the current value of $rnd[a]$ satisfies $rnd[a] \geq i$ for all a in Q. Let $vr(a)$ and $vv(a)$ be the values of $vrnd[a]$ and $vval[a]$, respectively, reported by acceptor a's round i phase 1b message, for a in Q. Let k be the largest value of $vr(a)$ for all acceptors a in Q. We now consider separately the two possible cases:

K1. $k = 0$

K2. $k > 0$

In case K1, every acceptor a in Q reported $vrnd[a] = 0$, so no acceptor in Q has voted in any round $j < i$. Since $rnd[a] \geq i$ for all a in Q, Observation 2 implies that no value has been or ever might be chosen in any round $j < i$. Hence, in this case $CP(v, i)$ is satisfied regardless of what value v the coordinator picks. However, to preserve nontriviality, the coordinator must pick a value v that has been proposed.

In case K2, one or more acceptors reported having voted in round k and no acceptor reported having voted in any round $j > k$. Let a_0 be some acceptor in Q with $vr(a_0) = k$. In phase 2a, the coordinator picks v to equal $vv(a_0)$. To show that this v satisfies $CP(v, i)$, we must show that for any round j with $j < i$, no value other than v has been or might yet be chosen in round j. Because $vrnd[a] \leq rnd[a]$ and acceptor a responds to a round i message only if $i > rnd[a]$, we must have $k < i$. The proof can therefore be split into the following three cases.

- $k < j < i$. Let a be any acceptor in Q. Because $vr(a)$ is the largest round number in which a had cast a vote when it sent that message and $vr(a) \leq k < j$, acceptor a had at that time not voted in round j. Since it set $rnd[a]$ to i upon sending the message, a could not subsequently have cast a vote in any round numbered less than i. Hence, a has not voted in round j. Therefore, no acceptor in Q has voted in round j. Since $rnd[a] \geq i > j$ for all a in Q, Observation 2 implies that no value has been or might yet be chosen in round j.

- $j = k$. Acceptor a_0 set $vrnd[a_0]$ to $k > 0$ and $vval[a_0]$ to v when it voted for v in round k. Since an acceptor can vote in a round only for the value sent it by the coordinator of that round, every acceptor has either voted for v or not voted in round k. Since $rnd[a] \geq i > k$ for all a in Q, Observation 3 implies that no value other than v has been or might yet be chosen in round j.

- $j < k$. We can assume by induction that property CP held when acceptor a_0 voted for v in round k. This implies that no value other than v has been or might yet be chosen in round j.

```
LET  $Q$  be any majority set of acceptors that have sent round $i$ phase 1b
          messages.
     $vr(a)$ and $vv(a)$ be the values of $vrnd[a]$ and $vval[a]$, respectively,
          reported by acceptor $a$'s phase 1b message, for $a$ in $Q$.
     $k$  be the largest value of $vr(a)$ for all $a$ in $Q$.
     $V$  be the set of values $vv(a)$ for all $a$ in $Q$ with $vr(a) = k$.
IF $k = 0$  THEN  choose $v$ to be any proposed value.
            ELSE  $V$ contains a single element; choose $v$ to be that element.
```

Figure 1: The coordinator's rule for picking value v in phase 2a of round i.

This completes the informal proof that $CP(v, i)$ is satisfied with $v = vv(a_0)$. A more rigorous proof, based on invariance, can be found in [7].

The rule for picking the value v in phase 2a is restated in Figure 1 in a somewhat different form that is more useful for developing Fast Paxos. Observe that the rule allows the coordinator to pick a v if it has received phase 1b messages from a majority set and it has received a proposal message from a proposer.

2.3 Progress

The simple algorithm of Section 2.2 satisfies property CP, and hence satisfies the consistency requirement. It obviously satisfies the nontriviality requirement. I now extend it to the complete algorithm that also satisfies progress.

2.3.1 The Progress Property

Paxos gets around the Fischer, Lynch, Paterson impossibility result by assuming an algorithm for selecting a single coordinator to be the *leader*. Of course, selecting a single leader is a form of consensus. However, Paxos requires a single leader only for achieving progress. Safety is preserved even if the leader-selection algorithm fails and the agents differ on what coordinator they believe to be the leader. In practice, it is usually not hard to implement a leader-selection algorithm that succeeds most of the time, which is good enough since nothing bad happens if the algorithm fails. I will not discuss how a leader is selected.

Since Paxos system model supportsageints stopping and restarting,"Non-faulty" DOES NOT mean always available. I t just means that agent performs all expected actions when needed. As long as that holds, the agent may permanently or temporarily fail whenever possible.

Lamport's conditions are high-level assumptions about the agents' future behavior that abstract the implementation details

Combined Fairness of send and receive,

An agent is defined to be *nonfaulty* iff it eventually performs the actions that it should, such as responding to messages. Define a set G of agents to be *good* iff all the agents in G are nonfaulty and, if any one agent in G repeatedly sends a message to any other agent in G, then that message is eventually received—more precisely, the message is eventually delivered or G is eventually not considered to be good. Being nonfaulty and being good are temporal properties that depend on future behavior.

By definition, any subset of a good set is also a good set. The union of two good sets need not be good if the communication network is partitioned, or if network connectivity is not transitively closed, so agents p and q may be unable to communicate with each other even though another agent r can communicate with both of them. If the network crashes completely, there will be no good sets containing more than a single agent.

For any proposer p, learner l, coordinator c, and set Q of acceptors, define $LA(p, l, c, Q)$ to be the condition that asserts:

$LA1$. $\{p, l, c\} \cup Q$ is a good set.

$LA2$. p has proposed a value.

$LA3$. c is the one and only coordinator that believes itself to be the leader.

The Paxos consensus algorithm satisfies the following property.

Progress For any learner l, if there ever exists proposer p, coordinator c, and majority set Q such that $LA(p, l, c, Q)$ holds from that time on, then eventually l learns a value.

This property asserts that l eventually learns a value if $LA(p, l, c, Q)$ holds forever, for suitable p, c, and Q. More useful would be a real-time property asserting that l learns a value within some specific length of time if $LA(p, l, c, Q)$ holds for that length of time. By assuming a time bound on how long it takes a message sent by any agent in a good set to be received and processed by another agent in that good set, it is straightforward to convert a proof of the progress property to a proof of such a real-time property.

Faulty agents may perform actions (though the assumption of non-Byzantine faults implies that they cannot perform incorrect actions). Therefore, progress requires that LA3 hold even for faulty agents. The state of a failed agent that never performs any further actions is obviously immaterial.

2.3.2 The Complete Algorithm

I now extend the algorithm described above so it satisfies the progress property. First, the actions of the coordinator and the acceptors are modified as follows:

CA1. If an acceptor a receives a phase 1a or 2a message for a round i with $i < rnd[a]$ and the coordinator of round $rnd[a]$ is not the coordinator of round i, then a sends the coordinator of round i a special message indicating that round $rnd[a]$ was begun. [If $i < rnd[a]$ and rounds i and $rnd[a]$ have the same coordinator, then the round i message is obsolete and is ignored.]

CA2. A coordinator c performs an action only if it believes itself to be the current leader. It begins a new round i only if either $crnd[c] = 0$ or it has learned that a round j has been started, for some j with $crnd[c] < j < i$.

Since any message can be lost, agents may have to retransmit messages to ensure they are eventually delivered. I modify the algorithm with:

Send is IOE

CA3. Each acceptor keeps resending the last phase 1b or 2b message it sent; any coordinator that believes itself to be the leader keeps resending to each acceptor the last phase 1a or 2a message that it sent; and each proposer that has proposed a value keeps resending that proposal to each coordinator.

CA3 requires agents to send an infinite sequence of messages. Section 2.4 explains how this unending retransmission of messages can be halted.

The actions described thus far state what actions an agent should try to perform. We cannot expect a failed agent to succeed in doing anything. All we can expect is:

CA4. A nonfaulty agent eventually performs any action that it should.

For example, CA3 and CA4 imply that while an acceptor a is nonfaulty, it must eventually resend the last phase 1b or 2b message it sent. CA4 does not forbid faulty agents from performing actions.

2.3.3 Proof of Progress

I now sketch the proof that the complete Paxos algorithm satisfies the progress property of Section 2.3.1. I assume that $LA(p, l, c, Q)$ holds from some time T_0 on, for proposer p, learner l, coordinator c, and majority set Q, and I show that l eventually learns a value. Let $LA1$–$LA3$ be the three conditions of $LA(p, l, c, Q)$.

1. From time T_0 on, no coordinator other than c executes any more actions.

   PROOF: By $LA3$ and CA2.

2. By some time $T_1 \geq T_0$, there is a round number i such that from T_1 on, $crnd[c] = i$.

   PROOF: By $LA1$, $LA3$, and CA4, c eventually starts executing a round. By step 1, no other coordinator starts a new round after T_0. Since only a finite number of rounds have been started by time T_0, eventually c will no longer learn of rounds with numbers greater than $crnd[c]$ and by CA2 will start no new round.

3. From T_1 on, $rnd[a] \leq i$ for all acceptors a in Q.

   PROOF: I assume $rnd[a] > i$ after T_1, for some a in Q, and obtain a contradiction. By step 2, LA1, LA3, and CA3, c sends an infinite sequence of round i messages to a. By $LA1$, CA1, and CA4, a sends an infinite number of messages to c informing it of round numbers greater than i. By $LA1$, $LA3$, and CA4, one of these messages is eventually received by c and causes it to start a round numbered greater than i, contradicting step 2.

4. By some time $T_2 \geq T_1$, coordinator c will have started phase 2 of round i.

   PROOF: Step 2 implies c has begun round i by time T_1. I assume it never starts phase 2 of round i and obtain a contradiction. By this assumption, step 2, $LA1$, $LA3$, and CA3, c must keep resending phase 1a messages. By $LA1$ and CA4, every acceptor in Q eventually receives this phase 1a message and, by step 3, responds (or has already responded) with a phase 1b message. By $LA1$, CA3, and step 1, the acceptors in Q keep resending those phase 1b messages and c eventually receives them. By $LA1$, $LA2$, and CA3, proposer p keeps sending a proposal message to c and c eventually receives it. When c has received the proposal and the phase 1b messages from the majority set Q, $LA3$ implies that its phase 2a action is enabled, so $LA1$ and CA4 imply that c performs this action, beginning phase 2 of round i. This is the required contradiction.

5. Eventually l learns a value.

Proof: By step 4, $LA1$, $LA3$, and CA3, c keeps sending phase 2a messages for round i with some single value v to the acceptors. By $LA1$, each acceptor in Q receives those messages and, by CA4 and step 2, it sends a round i phase 2b message with value v to l if it has not already done so. By CA3 and step 3, the acceptors in Q keep sending those phase 2b messages to l, so $LA1$ and CA4 imply l eventually learns v.

2.4 Implementation Considerations

We expect process failure and message loss to be rare events. (If physical messages are often lost, then an acknowledgement and retransmission protocol will be used to make logical message loss rare.) Almost always, no failure or message loss occurs between when a value is proposed and when it is chosen. I therefore consider implementation costs only in this normal case. However, it might be common for one or more agents to have already failed when a proposal is issued.

2.4.1 Reducing the Number of Messages

The algorithm described here never terminates and can keep sending unnecessary messages. I now describe some simple implementation techniques that can make it quite efficient.

First of all, there is no need to continue executing the algorithm once a value has been chosen. Any process that learns the chosen value can stop executing the algorithm and simply broadcast that value to all other processes.

CA3 requires that agents keep retransmitting messages that may already have been received. In some cases, the sender can tell that it is not necessary to keep sending the message. For example, a coordinator that has received a phase 1b message from an acceptor need no longer send it phase 1a messages for the same round—even though it has not yet sent a phase 2a message. In real systems, acknowledgements are often sent to tell the sender that a message has been received. However, if the receiver does not record the receipt of the message in stable storage, then the acknowledgement can serve only as a hint that the sender need not resend for a while. In practice, the appropriate use of acknowledgements and timeouts will avoid most retransmission if no message loss has occurred.

There are several ways of reducing the number of messages used by the algorithm. (These save messages in the normal case, but not necessarily in exceptional cases.)

- A coordinator can send a phase 1a or 2a message only to a majority of acceptors that it believes are nonfaulty. It can later send the message to other acceptors if one of the original acceptors does not respond.

- A proposer can send its proposal only to the leader rather than to all coordinators. However, this requires that the result of the leader-selection algorithm be broadcast to the proposers, which might be expensive. So, it might be better to let the proposer send its proposal to all coordinators. (In that case, only the coordinators themselves need to know who the leader is.)

- Instead of each acceptor sending phase 2b messages to each learner, acceptors can send their phase 2b messages to the leader and the leader can inform the learners when a value has been chosen. However, this adds an extra message delay.

Finally, observe that phase 1 is unnecessary for round 1, because a phase 1b message sent by an acceptor a in that round can report only $vrnd[a] = 0$. (Acceptor a will not send the phase 1b message for round 1 if $rnd[a] \geq 1$.) The coordinator of round 1 can begin the round by sending a phase 2a message with any proposed value.

2.4.2 The Cost of Classic Paxos

The efficiency of a consensus algorithm would not matter if a system were to execute it only once. In most applications, the system executes a sequence of instances of the algorithm with the same agents, using the same leader for each instance [7, Section 3], [9, Section 3].

Key to the efficiency of the Paxos consensus algorithm is the observation that phase 1 can be performed simultaneously for all instances. When a new leader c is selected, either initially or because the previous leader failed, c chooses a round number i for which it is coordinator that it believes is larger than that of any previously started round. It then sends round i phase 1a messages for all instances whose outcome it does not already know. An acceptor responds with phase 1b messages for all those instances. As explained elsewhere [7, 9, 12], those messages contain only a small amount of information, and the messages sent by each agent for all those instances are easily combined into a single physical message. The amortized cost of phase 1 is therefore negligible, and only the cost of phase 2 matters.

In almost all instances, every acceptor a reports $vrnd[a] = 0$. The coordinator then waits for a proposal message and sends its proposed value in its phase 2a message. In the normal case, the Paxos algorithm then works as follows, when we use the techniques for eliminating messages described in Section 2.4.1 above.

- The proposer sends a message either to the leader or to all coordinators.
- The leader sends phase 2a messages to a majority set of acceptors.
- The majority set sends phase 2b messages to the learners.

There are three potentially significant costs to executing the Paxos consensus algorithm. The two obvious ones are the latency, measured in message delays, and the communication bandwidth, measured in number of messages. The latency is three message delays. The number of messages depends on implementation details such as the number of coordinators to which a proposal is sent. Let N be the number of acceptors, and suppose phase 1a and 2a messages are sent only to a majority set. With multicast, so a single message can have multiple recipients, a total of $\lfloor N/2 \rfloor + 3$ messages are required. Suppose proposals are sent only to the leader, which is an acceptor, and the acceptors are the learners (a common situation for client/server systems). With unicast (point-to-point messages), $N(\lfloor N/2 \rfloor + 1)$ messages are sent.

The less obvious cost is the latency caused by writing information to stable storage. As observed in Section 2.1, allowing a failed agent to be restarted requires that its state be recorded in stable storage. For many systems, writing to stable storage can be much more expensive than sending a message. Thus, in executing phase 2, the leader must perform a write to stable storage before sending its phase 2a messages, and acceptors must write to stable storage before sending their phase 2b messages. Since the latter writes are concurrent, this yields a latency of two stable-storage writes.

Whether proposers and learners must write to stable storage depends on the application. A learner can always ask the current leader what value, if any, has been chosen for a particular instance of the consensus algorithm. If the leader does not know, perhaps because it was just selected to be leader, it can start a new round to find out. It will either discover in phase 1 that no value has been chosen or else will execute phase 2a for a value that might already have been chosen.

3 Making Paxos Fast

As explained in Section 2.4.2 above, the normal-case communication pattern in the Paxos consensus algorithm is:

proposer $\rightarrow$ leader $\rightarrow$ acceptors $\rightarrow$ learners

In Fast Paxos, the proposer sends its proposal directly to the acceptors, bypassing the leader. This can save one message delay (and one message). I now explain how it works. But first, I need to generalize the Paxos algorithm in a small but important way.

The Paxos algorithm is stated above in terms of majority sets, where a majority set comprises a majority of the acceptors. A value v is chosen in round i iff a majority set of acceptors vote to accept v in round i. The only property required of majority sets is that any two majority sets have non-empty intersection. The algorithm trivially generalizes by assuming an arbitrary collection of sets called *quorums* such that any two quorums have non-empty intersection, and simply replacing "majority set" by "quorum" throughout Section 2.

We can further generalize the algorithm by allowing the set of quorums to depend on the round number. That is, we assume, for each round number i, a set of sets of acceptors called *i-quorums*. A value v is chosen in round i iff all acceptors in some i-quorum vote to accept v in round i. Again, the only requirement needed to preserve consistency is that any two quorums have non-empty intersection. This means that for any round numbers i and j, any i-quorum and any j-quorum have non-empty intersection. The necessary changes to the algorithm are obvious. For example, in Figure 1, instead of being a majority set, Q should be an i-quorum. I will not bother explicitly rewriting the algorithm.

It is not obvious what "majority set" should be replaced with in the progress property of Section 2.3. This is discussed in Section 3.3 below.

3.1 The Basic Algorithm

I now describe the generalization to Fast Paxos of the basic Paxos algorithm of Section 2.2. It guarantees only safety. Progress and cost are considered later.

In Fast Paxos, round numbers are partitioned into *fast* and *classic* round numbers. A round is said to be either fast or classic, depending on its number. Rounds proceed in two phases, just as before, except with two differences:

- The rule by which the coordinator picks a value in phase 2a is modified as explained below.

- In a fast round i, if the coordinator can pick any proposed value in phase 2a, then instead of picking a single value, it may instead send a special phase 2a message called an *any* message to the acceptors. When an acceptor receives a phase 2a *any* message, it may treat any proposer's message proposing a value as if it were an ordinary round i phase 2a message with that value. (However, it may execute its round i phase 2b action only once, for a single value.)

Recall that, in the normal case, all phase 1b messages received by the coordinator c indicate that no acceptors have voted in that round, so c may pick any proposed value in phase 2a. Therefore, the normal case for a fast round is for c to send a phase 2a *any* message. This is normally done when c is selected to be leader, before any values are proposed for that instance of the algorithm. An acceptor then waits for a proposal message from a proposer, and it treats the first one it receives as if it were an ordinary phase 2a message from c.

A classic round works the same as in classic Paxos. The coordinator picks the value that the acceptors can vote for, so different acceptors cannot vote to accept different values in the same round. This is not the case in a fast round. If the coordinator sends a phase 2a *any* message in a fast round, each acceptor independently decides what proposal message to take as a phase 2a message. Different acceptors can therefore vote to accept different values in a fast round.

The coordinator's rule for picking a value in phase 2a no longer guarantees consistency, even for a classic round. In fact, that rule no longer works at all. The rule's statement in Figure 1 asserts that, if $k \neq 0$, then V contains a single element. This is no longer true if k is a fast round number. The rule for picking a value v in phase 2a must be revised.

To discover how to revise the rule, we must review its derivation in Section 2.2.2. We want to maintain property CP, which means that $CP(v, i)$ must hold for the value v picked in round i. Observations 1–3 remain valid, with *majority set* replaced by j*-quorum*.

The reasoning is the same as before for case K1, in which $k = 0$. In this case, $CP(v, i)$ holds for all values v. The coordinator therefore can send a phase 2a message with any proposed value, or a phase 2a *any* message for a fast round.

We now consider case K2, $k > 0$. Let V and Q be defined as in Figure 1. We must find a value v in V that satisfies $CP(v, i)$. This means showing that no value other than v has been or might yet be chosen in round j, for all $j < i$. In the reasoning for classic Paxos in Section 2.2.2, we consider the three possible cases $k < j < i$, $j = k$, and $j < k$. We used the fact that V contains at most one value only in the $j = k$ case . Therefore, the same reasoning as before for these cases shows:

- $k < j < i$. No value has been or might yet be chosen in round j.
- $j = k$. If V contains the single value v, then no value other than v has been or might yet be chosen in round j.
- $j < k$. For any v in V, no value other than v has been or might yet be chosen in round j.

This takes care of everything, except the case in which $j = k$ and V contains more than one value—a cast that is possible only if k is a fast round number. In this case, no value has been or might be chosen in any round numbered less than i except possibly in round k. (For $j < k$, this follows because if no value other than v can be chosen in round j for two different values v, then no value can be chosen in round j.) To handle the case of $j = k$ and V having more than one value, we must reason more carefully.

By Observation 1, a value v might have been or might yet be chosen in round k only if there is a k-quorum R such that every acceptor a in R has $rnd[a] \leq k$ or has voted for v in round k. Since every acceptor a in Q has $rnd[a] \geq i > k$ (because a has sent a round i phase 1b message), this implies:

> **Observation 4.** With Q, vr, vv, and k as in Figure 1, a value v has been or might yet be chosen in round k only if there exists a k-quorum R such that $vr(a) = k$ and $vv(a) = v$ for every acceptor a in $R \cap Q$.

Define $O4(v)$ to be true iff there exists a k-quorum R such that $vr(a) = k$ and $vv(a) = v$ for all acceptors a in $R \cap Q$. Observation 4 asserts that v has been or might be chosen in round k only if $O4(v)$ is true. There are three cases to consider:

1. There is no v in V satisfying $O4(v)$. In this case, Observation 4 implies that no value has been or might yet be chosen in round k. Hence, the coordinator can pick any value $v \in V$ in phase 2a.
2. There is a single v in V satisfying $O4(v)$. In this case, Observation 4 implies that v is the only value that has been or might yet be chosen in round k. Hence, the coordinator can pick v in phase 2a.

3. There is more than one v in V satisfying $O4(v)$. In this case we are stuck.

The solution to the dilemma of case 3 is to make that case impossible. Case 3 asserts that $O4(v)$ and $O4(w)$ are true for distinct values v and w. This implies that there are k-quorums R_v and R_w such that $vv(a) = v$ for all a in $R_v \cap Q$ and $vv(a) = w$ for all a in $R_w \cap Q$. This is impossible if $R_v \cap R_w \cap Q$ is nonempty. We already require that any two quorums have non-empty intersection. We make case 3 impossible by strengthening this to:

Quorum Requirement For any round numbers i and j:

(a) Any i-quorum and any j-quorum have non-empty intersection.

(b) If j is a fast round number, then any i-quorum and any two j-quorums have non-empty intersection.

Section 3.3 below discusses how this requirement is satisfied.

This completes the derivation of the rule for choosing the value that the coordinator sends in its phase 2a messages for round i. The derivation shows that, with this rule, Fast Paxos satisfies the consistency requirement for consensus. The rule is summarized in Figure 2.

If i is a fast round number, then in the first THEN case or the final ELSE case of Figure 2, the coordinator could send a phase 2a *any* message. However, it is better for it to send a proposed value if it knows one, since that avoids the possibility of collision discussed below. In the final ELSE case, the coordinator knows that every value in V has been proposed, so it should send a proposed value.

3.2 Collision Recovery

In classic Paxos, a round i succeeds in choosing a value if an i-quorum of acceptors receive that round's phase 2a message before receiving a message for a higher-numbered round. This is not true for Fast Paxos if i is a fast round in which the coordinator sends a phase 2a *any* message. In that case, different acceptors can vote to accept different values in that round, resulting in no value being chosen. In the normal case, an acceptor first receives the phase 2a *any* message and then votes to accept the first proposed value it receives. The round can fail if two or more different proposers send proposals at about the same time, and those proposals are received by the acceptors in different orders. I now consider what the algorithm does to recover from such a collision of competing proposals.

```
LET  $Q$ be any $i$-quorum of acceptors that have sent round $i$ phase 1b
         messages.
     $vr(a)$ and $vv(a)$ be the values of $vrnd[a]$ and $vval[a]$, respectively,
         reported by acceptor $a$'s phase 1b message, for $a$ in $Q$.
     $k$ be the largest value of $vr(a)$ for all $a$ in $Q$.
     $V$ be the set of values $vv(a)$ for all $a$ in $Q$ with $vr(a) = k$.
     $O4(v)$ be true iff there is a $k$-quorum $R$ such that $vr(a) = k$ and
         $vv(a) = v$ for all $a$ in $R \cap Q$.
IF $k = 0$ THEN let $v$ be any proposed value.
           ELSE IF $V$ contains a single element
                  THEN let $v$ equal that element.
                  ELSE IF there is some $w$ in $V$ satisfying $O4(w)$
                         THEN let $v$ equal that $w$ (which is unique)
                         ELSE let $v$ be any proposed value
```

Figure 2: The coordinator's rule for picking a v that it can send in its phase 2a message for round i.

The obvious way to recover from a collision is for c to begin a new round, sending phase 1a messages to all acceptors, if it learns that round i may not have chosen a value. Suppose the coordinator c of round i is also coordinator of round $i+1$, and that round $i+1$ is the new one it starts. The phase 1b message that an acceptor a sends in response to c's round $i+1$ phase 1a message does two things: it reports the current values of $vrnd[a]$ and $vval[a]$, and it transmits a's promise not to cast any further vote in any round numbered less than $i+1$. (This promise is implicit in a's setting $rnd[a]$ to $i+1$.) Suppose a voted in round i. In that case, a's phase 1b message reports that $vrnd[a] = i$ and that $vval[a]$ equals the value a sent in its round i phase 2b message. Moreover, that phase 2b message also implies that a can cast no further vote in any round numbered less than $i+1$. In other words, a's round i phase 2b message carries exactly the same information as its round $i+1$ phase 1b message. If coordinator c has received the phase 2b message, it has no need of the phase 1b message. This observation leads to the following two methods for recovery from collision.

The first method is *coordinated* recovery. Suppose i is a fast round number and c is coordinator of rounds i and $i+1$. In coordinated recovery, acceptors send their round i phase 2b messages to the coordinator c (as well as to the learners). If c receives those messages from an $(i+1)$-quorum of acceptors and sees that a collision may have occurred, then it treats those phase 2b messages as if they were the corresponding round $i+1$ phase 1b messages and executes phase 2a of round $i+1$, using the rule of Figure 2 to choose the value v in its phase 2a messages. (Note that an acceptor can perform its phase 2b action even if it never received a phase 1a message for the round.) Since V is non-empty in this case, c does not send a phase 2a *any* message. Hence, round $i+1$ will succeed if the acceptors in a nonfaulty $(i+1)$-quorum receive those phase 2a messages before receiving any message from a higher-numbered round. Coordinated recovery is forbidden by modification CA2 (Section 2.3.2), which was added to guarantee progress. CA2 is amended below to correct this problem.

The second method is *uncoordinated* recovery. Suppose i and $i+1$ are both fast round numbers. In uncoordinated recovery, acceptors send their round i phase 2b messages to all other acceptors. Each acceptor uses the same procedure as in coordinated recovery to pick a value v that the coordinator could send in a round $i+1$ phase 2a message. It then executes phase 2b for round $i+1$ as if it had received such a phase 2a message. Because of nondeterminism in the rule for picking v, different acceptors could vote to accept different values in round $i+1$, preventing the round from succeeding. However, since $i+1$ is a fast round number, consistency is preserved and a higher-numbered round can still choose a value. Section 3.4 discusses ways of trying to get all the acceptors to pick the same v.

Coordinated and uncoordinated recovery add new allowed actions to the algorithm. These actions maintain safety because they are equivalent to actions allowed by the original algorithm. A coordinator that learns of a collision in round i can still start a complete new round with a number greater than $i+1$. (Like coordinated recovery, this will be allowed by the amended version of CA2.)

3.3 Progress

To specify the progress property that Fast Paxos must solve, we have to find the appropriate replacement for *majority set* in the classic Paxos progress property of Section 2.3. The obvious replacement is i-*quorum*, but for what i? Answering that question requires considering how quorums are selected.

The Quorum Requirement of Section 3.1 states requirements for i-quorums that differ depending on whether i is a classic or fast round number.

There seems to be no other reason why one would want the set of i-quorums to depend on i. I therefore assume two types of quorums—classic and fast. An i-quorum is a classic or fast quorum depending on whether i is a classic or fast round number. The Quorum Requirement then asserts that (a) any two quorums have non-empty intersection and (b) any two fast quorums and any classic or fast quorum have a non-empty intersection.

To obtain the progress property for Fast Paxos, we are led to replace *majority set* in the classic Paxos progress property with either *classic quorum* or *fast quorum*. But which should we choose? Since the requirements for fast quorums are stronger than for classic quorums, fast quorums will be at least as large as classic ones. So, requiring only a classic quorum to be nonfaulty for progress gives a stronger progress property than requiring a fast quorum to be nonfaulty. I will now enhance the Fast Paxos algorithm to make it satisfy the stronger property, obtained by substituting *classic quorum* for *majority set* in the classic Paxos progress property.

If i is a classic round number, then round i works exactly the same in Fast Paxos as in Classic Paxos. If we ensure that a leader eventually starts a round with a large enough classic round number, then the same argument as in Section 2.3.3 above shows that Fast Paxos satisfies its progress property. (Since uncoordinated recovery allows acceptors executing round i to start a new round $i+1$, the proof of the progress property requires the additional observation that this can happen only if i is a fast round number.) To ensure that the leader can start a classic round with a large enough number, we make the assumption that each coordinator is the coordinator for infinitely many classic round numbers. (This is a condition on the pre-assigned mapping from rounds to coordinators.) We also modify CA2 to:

CA2′. A coordinator c may perform an action only if it believes itself to be the current leader. It may begin a new round i only if (a) $crnd[c] = 0$, (b) $crnd[c]$ equals a fast round number and i is a classic round number, or (c) it has learned that a round j has been started, for some j with $crnd[c] < j < i$.

Case (b) allows coordinated recovery from a collision in a fast round i if $i+1$ is a classic round number, as well as starting a completely new classic round. Since the leader never sends a phase 2a *any* message in coordinated recovery, and classic quorums are at least as small as fast ones, there is no reason to perform coordinated recovery with $i+1$ a fast round. CA4 must be interpreted to mean that if coordinator c believes itself to be the leader and $crnd[c]$ remains a fast round number, then c must eventually begin a new classic round. As I have observed, this ensures that Fast Paxos satisfies its liveness property.

3.4 Implementation Considerations

3.4.1 Choosing Quorums

In most applications, whether a set of acceptors forms a quorum depends only on its cardinality. Let N be the number of acceptors, and let us choose F and E such that any set of at least $N - F$ acceptors is a classic quorum and any set of at least $N - E$ acceptors is a fast quorum. This means that a classic round can succeed if up to F acceptors have failed, and a fast round can succeed if up to E acceptors have failed. We would like E and F to be as large as possible, though increasing one may require decreasing the other.

Recall that the Quorum Requirement asserts that (a) any two quorums have non-empty intersection and (b) any two fast quorums and any classic or fast quorum have a non-empty intersection. Since the requirements for fast quorums are more stringent than for classic quorums, we can always assume $E \leq F$. (If $F < E$, we could replace F by E and still satisfy the requirements.) It is not hard to show that $E \leq F$ implies that (a) and (b) are equivalent to:

(a) $N > 2F$

(b) $N > 2E + F$

For a fixed N, the two natural ways to choose E and F are to maximize one or the other. Maximizing E under the constraints (a) and (b) yields $E = F = \lceil N/3 \rceil - 1$; maximizing F under those constraints yields $F = \lceil N/2 \rceil - 1$ and $E = \lfloor N/4 \rfloor$.

With $E < F$, the leader can decide whether to employ a fast or slow round number based on the number of acceptors that are nonfaulty. If there are at least $N - E$ nonfaulty acceptors, it can use a fast round. If more than E acceptors have failed, then the leader can switch to classic Paxos by beginning a new round with a classic round number, executing phase 1 for all instances. (See Section 2.4.2.)

3.4.2 Avoiding Collisions in Uncoordinated Recovery

In uncoordinated recovery, an acceptor picks a value to vote for in round $i + 1$ based on the round i phase 2b messages it receives. It uses the rule of Figure 2, treating those phase 2b messages as round $i + 1$ phase 1b messages. The nondeterminism in that rule could lead different acceptors to pick different values, which could prevent round $i + 1$ from choosing a value. I now show how to prevent that possibility.

For a fixed $(i + 1)$-quorum Q, the nondeterminism is easily removed by using a fixed procedure for picking a value from V in the final ELSE clause of Figure 2. However, different acceptors could use different $(i+1)$-quorums Q if they do not all receive the same set of round i phase 2b messages. This can be prevented by having the round's coordinator indicate in its round i phase 2a *any* message what $(i + 1)$-quorum Q should be used for uncoordinated recovery in case of collision. (It can select as Q any quorum it believes to be nonfaulty.) The coordinator could also indicate Q implicitly, by first sending its phase 2a message only to a minimal fast quorum Q of nonfaulty acceptors, as described in Section 2.4. In the normal case when none of the acceptors in Q fail, then the set of round i phase 2b messages received by the acceptors in Q will be exactly the ones sent by the acceptors in Q, so all the acceptors in Q will use Q as the $(i + 1)$-quorum when picking a value for their round $i + 1$ phase 2b messages. Round $i + 1$ will then succeed.

3.4.3 The Cost of Fast Paxos

As explained in Section 2.4.2, a newly selected leader selects a new round number and executes phase 1 concurrently for multiple instances of the consensus algorithm. If $E < F$, so fast quorums are larger than classic quorums, the leader will probably choose a fast round number iff it believes that a fast quorum of acceptors is nonfaulty. If $E = F$, in most applications it will always select a fast round number. If the leader chooses a classic round number, then everything works as in classic Paxos. The only difference in cost is that a quorum may consist of more than a majority of acceptors, so more messages are sent. Suppose now that it chooses a fast round number.

For almost all instances, the leader finds in phase 1 that it can use any proposed value. It therefore sends a phase 2a *any* message to all acceptors in some quorum. Since this message is sent for almost all instances, only a single physical message is needed for all those instances and the amortized cost is negligible. The significant cost of Fast Paxos begins with the proposer's proposal message.

In classic Paxos, proposers can be informed of who the leader is, and they can send their proposals to just the leader. However, it might be easier to have them send proposals to all coordinators rather than informing them who the current leader is. Similarly, in Fast Paxos, proposers can learn that fast rounds are being executed and that the leader has chosen some specific nonfaulty quorum of acceptors to whom they should send their proposals. However, it might be easier to have them send their proposals to all coordinators and acceptors.

In the absence of collision, the round proceeds exactly as in classic Paxos, except with one fewer message delay and with somewhat more messages, since a fast quorum generally contains more than a majority of acceptors. Moreover, the use of coordinated or uncoordinated recovery from collision requires that phase 2b messages be sent to the leader (coordinated recovery) or to a fast quorum of acceptors (uncoordinated recovery). This may require additional unicast (but not multicast) messages, depending on whether the leader or the acceptors are also learners. For comparison with classic Paxos, let us again assume that the leader is an acceptor, the acceptors are the learners, and phase 2a messages are sent only to a quorum. Let us also assume that $E = F$ and proposals are sent only to a quorum. Under these assumptions, $\lfloor 2N/3 \rfloor + 2$ multicast or $N(\lfloor 2N/3 \rfloor + 1)$ unicast messages are required.

If there is a collision in a fast round, then the algorithm incurs the additional cost of recovery. That cost depends on how recovery is performed—with a completely new round, with coordinated recovery, or with uncoordinated recovery. The coordinator will decide in advance which type of recovery should be used. Its decision can be either implied by the round number or announced in the phase 2a *any* messages.

Starting a new round incurs the cost of phases 1 and 2 of the new round. The cost of coordinated recovery from a collision in round i is the cost of phase 2 of round $i+1$, which includes the latency of two message delays and of two writes to stable storage. With the assumptions above, this means an additional $\lfloor 2N/3 \rfloor + 2$ multicast or $N(\lfloor 2N/3 \rfloor + 1)$ unicast messages. Uncoordinated recovery adds only the cost of phase 2b of round $i + 1$, including the latency of one message delay and of one write to stable storage. With the same assumptions, this gives $\lfloor 2N/3 \rfloor + 1$ multicast or $(N - 1)(\lfloor 2N/3 \rfloor + 1)$ unicast messages. This makes uncoordinated recovery seem better than coordinated recover, which in turn seems better than starting a complete new round. However, these numbers assume that acceptors are learners. If not, uncoordinated recovery may add more phase 2b messages to round i than does coordinated recovery (even in the absence of collision); and recovery by starting a complete new round adds no extra messages to round i.

The best method of recovering from a collision depends on the cost of sending those extra round i messages and on how often collisions occur. With multicast, there is probably no cost to those extra messages, in which case uncoordinated recovery is best. If collisions are very rare, then starting a new round might be best. If collisions are too frequent, then classic Paxos might be better than Fast Paxos.

Remember that a collision does not occur just because two proposals are sent at the same time; it requires those proposals to be received in different order by different acceptors. How often that happens depends on how often proposals are generated and on the nature of the communication medium. For example, if all the acceptors are on the same Ethernet, then it is very unlikely for messages to be received in different order by different acceptors, so collisions will be extremely rare.

4 Conclusion

The consensus problem is best expressed in terms of proposers that propose values, acceptors that choose a value, and learners that learn the chosen value. Traditional asynchronous consensus algorithms require three message delays between proposal and learning. In normal operation, Fast Paxos requires only two message delays in the absence of collision: a proposer sends its proposal to acceptors and the acceptors send phase 2b messages to the learners. If a collision does occur, uncoordinated recovery normally adds only one message delay: the acceptors send another round of phase 2b messages to the learners. These are the minimum message delays required by a general consensus algorithm—that is, one that works for arbitrary sets of proposers, acceptors, and learners [11].

The number of nonfaulty acceptors required to ensure progress is determined by the Quorum Requirement. The Fast-Accepting Lemma in Section 2.3 of [11] shows that this requirement must be satisfied by any general fast consensus algorithm. Quorums can be chosen so progress with fast learning is ensured if more than $2/3$ of the acceptors are nonfaulty. Alternatively, they can be chosen so progress is ensured if a majority of the acceptors are nonfaulty and fast learning occurs if at least $3/4$ of the acceptors are nonfaulty.

The algorithm of Brasileiro et al. [1] can be converted to a fast consensus algorithm by having each of their processes use as its private value the first proposed value it receives. Their N processes are then the acceptors and learners. (It is easy to add other learners.) The algorithm starts with what is essentially a fast round 1 of a version of Fast Paxos in which fast and classic quorums both contain $N - F$ acceptors, where $N > 3F$. If that round does not succeed, the algorithm switches to an ordinary consensus algorithm where each process takes as its private value a value that it could send in a phase 2b message for round 2 with uncoordinated recovery. This works because Fast Paxos satisfies property CP (Section 2.2.2), so if a value v has been or might yet be chosen by round 1, then no process can use any value other than v as its private value. The R-Consensus Algorithm of Pedone and Schiper [15] is similar to a version of Fast Paxos in which all rounds are fast.

The One-Two Consensus Algorithm of Zielinski [16] concurrently executes a fast and classic protocol for each round. If a collision prevents the fast protocol from choosing a value in two message delays, then the classic protocol can choose the value in one more message delay. Consistency is maintained by allowing the fast protocol to choose only a value that is proposed by the leader in the classic protocol. Failure of a single process (the leader) makes fast learning impossible, so this is formally an $E = 0$ protocol. However, it allows fast learning despite the failure of $E > 0$ acceptors different from the leader.

Both classic and Fast Paxos can be generalized to handle Byzantine (malicious) failures. The Castro-Liskov algorithm [2] is one version of classic Byzantine Paxos. In the normal case, this algorithm requires one more message delay than ordinary classic Paxos, for a total of four message delays between proposal and learning. The extra message delay is needed to prevent a malicious coordinator from sending phase 2a messages with different values. Since Fast Paxos allows phase 2a messages with different values in fast rounds, it can be generalized to a faster Byzantine Paxos algorithm that eliminates the extra message delay. The Fast Byzantine Consensus algorithm developed independently by Martin and Alvisi [13] is one version of the resulting algorithm. Fast Paxos can also be generalized to a Fast Byzantine Paxos algorithm that requires only two message delays between proposal and learning in the absence of collisions. (However, a single malicious proposer can by itself create a collision.) Descriptions of these generalizations of classic and Fast Paxos will appear elsewhere.

Acknowledgment

I wish to thank Fernando Pedone for his comments on an earlier version.

References

[1] Francisco Brasileiro, Fabíola Greve, Achour Mostefaoui, and Michel Raynal. Consensus in one communication step. In V. Malyshkin, editor,

Parallel Computing Technologies (6th International Conference, PaCT 2001), volume 2127 of *Lecture Notes in Computer Science*, pages 42–50. Springer-Verlag, 2001.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. ACM, 1999.

[3] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[4] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243:35–91, 2000.

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

[6] Leslie Lamport. Introduction to TLA. SRC Technical Note 1994-001, Digital Systems Research Center, December 1994. Currently available from `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.html`.

[7] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[8] Leslie Lamport. A summary of TLA$^+$. Currently available from `http://research.microsoft.com/users/lamport/tla/tla.html` or by searching the Web for the 21-letter string obtained by removing the - characters from uid-lamport-tla-homepage, 2000.

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

[10] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.

[11] Leslie Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-71, Microsoft Research, July 2004. Currently available from `http://research.microsoft.com/users/lamport/pubs/pubs.html`, or by searching the Web for the 23-letter

string obtained by removing the - characters from all-lamports-pubs-onthe-web.

[12] Butler W. Lampson. How to build a highly available system using consensus. In Ozalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, 1996. Springer-Verlag.

[13] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, Yokohama, June 2005. IEEE Computer Society. To appear.

[14] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast. *Distributed Computing*, 15(2):97–107, 2002.

[15] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference (EDCC-4)*, volume 2485 of *Lecture Notes in Computer Science*, pages 44–61. Springer-Verlag, 2002.

[16] Piotr Zielinski. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *DISC '05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, 2005.

Appendix: The Formal Specification

I now present a formal TLA$^+$ [10] specification of Fast Paxos. What such a specification means is explained briefly in [6]. The TLA$^+$ notation is summarized in the first four pages of [8].

For simplicity, the specification does not describe proposers and learners. Instead, it contains a variable *proposed* whose value represents the set of proposed values and a variable *learned* whose value represents the set of learned values. An action that in an implementation would be enabled by the receipt of a message proposing a value v is instead enabled by v being an element of *proposed*. A value is added to the set *learned* when *sentMsg* contains a set of messages that would enable a learner to learn that value.

The leader-selection algorithm is represented by a variable *amLeader*, where $amLeader[c]$ is a Boolean that is true iff coordinator c believes itself to be the current leader.

The specification contains a variable *goodSet* whose value is a set of acceptors and coordinators that is assumed to be a good set. (The specification does not rule out the possibility that there are other good sets as well.) Condition CA4, which requires nonfaulty agents to perform their actions, is formalized by weak fairness requirements on actions of agents in *goodSet*. The specification's requirement is actually weaker than CA4, since it applies only to nonfaulty agents that are in *goodSet*.

The specification describes message passing in terms of a variable *sentMsg* that represents the set of all messages present in the communication medium. Sending a message is represented by adding it to *sentMsg*, losing the message is represented by removing it from *sentMsg*. An operation that in an implementation would occur upon receiving some set of messages is represented by an action that is enabled when that set of messages is in *sentMsg*. The action does not remove those messages from *sentMsg*, so the specification allows the same message to be "received" multiple times.

The algorithm requires an agent to keep retransmitting its most recent message. We let the set *sentMsg* include that message. We can think of the buffer in which the agent keeps that message as part of the communication medium represented by *sentMsg*, so the message can be lost (removed from *sentMsg*) only if the agent fails. The specification therefore does not allow the last message sent by an agent in *goodSet* to be lost. Since the presence of a message in *sentMsg* allows the message to be "received", there is no need for an explicit action to keep retransmitting it. (However, the specification does include explicit retransmission actions in case an agent has failed and been repaired after its last message has been lost.)

Because messages are not removed from *sentMsg* when they are received, the specification can be simplified by allowing a single element of *sentMsg* to represent messages sent to multiple agents. A message therefore does not specify its recipient. A coordinator sends a single phase 1a or 2a message (by adding it to the set *sentMsg*) that can be received by any acceptors. Moreover, a round j message sent by an acceptor to the coordinator of that round can be received by other coordinators as well. The specification is further simplified by eliminating the messages that CA1 (Section 2.3.2) requires an acceptor a to send when it receives a round i message when $i < rnd[a]$. The message sent by a for round $rnd[a]$ to the coordinator of that round serves to notify the coordinator of round i that round $rnd[a]$ has been started.

MODULE *FastPaxos*

The module imports two standard modules. Module *Naturals* defines the set *Nat* of naturals and the ordinary arithmetic operators; module *FiniteSets* defines *IsFiniteSet*(S) to be true iff S is a finite set and defines *Cardinality*(S) to be the number of elements in S, if S is finite.

EXTENDS *Naturals*, *FiniteSets*

Constants

Max(S) is defined to be the maximum of a nonempty finite set S of numbers.

$$Max(S) \triangleq \text{CHOOSE } i \in S : \forall j \in S : j \leq i$$

The next statement declares the specification's constant parameters, which have the following meanings:

Val	the set of values that may be proposed.
Acceptor	the set of acceptors.
FastNum	the set of fast round numbers.
Quorum(i)	the set of i-quorums.
Coord	the set of coordinators.
Coord(i)	the coordinator of round i.

CONSTANTS *Val*, *Acceptor*, *FastNum*, *Quorum*(_), *Coord*, *CoordOf*(_)

RNum is defined to be the set of positive integers, which is the set of round numbers.

$$RNum \triangleq Nat \setminus \{0\}$$

The following statement asserts the assumption that *FastNum* is a set of round numbers.

ASSUME *FastNum* $\subseteq$ *RNum*

ClassicNum is defined to be the set of classic round numbers.

$$ClassicNum \triangleq RNum \setminus FastNum$$

The following assumption asserts that the set of acceptors is finite. It is needed to ensure progress.

ASSUME *IsFiniteSet*(*Acceptor*)

The following asserts the assumptions that *Quorum*(i) is a set of sets of acceptors, for every round number i, and that the Quorum Requirement (Section 3.1, page 19) holds.

```
ASSUME $\forall i \in RNum :$
          $\land\ Quorum(i) \subseteq \text{SUBSET } Acceptor$
          $\land\ \forall j \in RNum :$
               $\land\ \forall Q \in Quorum(i), R \in Quorum(j) : Q \cap R \neq \{\}$
               $\land\ (j \in FastNum) \Rightarrow$
                    $\forall Q \in Quorum(i) : \forall R1, R2 \in Quorum(j) :$
                      $Q \cap R1 \cap R2 \neq \{\}$
```

The following asserts the assumptions that $CoordOf(i)$ is a coordinator, for every round number i, and that every coordinator is the coordinator of infinitely many classic rounds.

$$\begin{aligned}
&\text{ASSUME } \land \forall i \in RNum : CoordOf(i) \in Coord \\
&\qquad\quad\ \ \land \forall c \in Coord, i \in Nat : \\
&\qquad\qquad\quad \exists j \in ClassicNum : (j > i) \land (c = CoordOf(j))
\end{aligned}$$

any and *none* are defined to be arbitrary, distinct values that are not elements of *Val*.

$$\begin{aligned}
any &\triangleq \text{CHOOSE } v : v \notin Val \\
none &\triangleq \text{CHOOSE } n : n \notin Val \cup \{any\}
\end{aligned}$$

Message is defined to be the set of all possible messages. A message is a record having a *type* field indicating what phase message it is, a *rnd* field indicating the round number. What other fields, if any, a message has depends on its type.

$$\begin{aligned}
&Message \triangleq \\
&\qquad\ \ [type : \{\text{"phase1a"}\}, rnd : RNum] \\
&\quad \cup\ [type : \{\text{"phase1b"}\}, rnd : RNum, vrnd : RNum \cup \{0\}, \\
&\qquad\quad vval : Val \cup \{any\}, acc : Acceptor] \\
&\quad \cup\ [type : \{\text{"phase2a"}\}, rnd : RNum, val : Val \cup \{any\}] \\
&\quad \cup\ [type : \{\text{"phase2b"}\}, rnd : RNum, val : Val, acc : Acceptor]
\end{aligned}$$

Variables and State Predicates

The following statement declares the specification's variables, which have all been described above—either in Section 2.2.1 on page 4 or in this appendix.

$$\begin{aligned}
&\text{VARIABLES } rnd, vrnd, vval, crnd, cval, amLeader, sentMsg, proposed, \\
&\qquad\qquad\quad learned, goodSet
\end{aligned}$$

Defining the following tuples of variables makes it more convenient to state which variables are left unchanged by the actions.

$aVars \triangleq \langle rnd, vrnd, vval \rangle$	Acceptor variables.
$cVars \triangleq \langle crnd, cval \rangle$	Coordinator variables.
$oVars \triangleq \langle amLeader, proposed, learned, goodSet \rangle$	Most other variables.
$vars \triangleq \langle aVars, cVars, oVars, sentMsg \rangle$	All variables.

TypeOK is the type-correctness invariant, asserting that the value of each variable is an element of the proper set (its "type"). Type correctness of the specification means that *TypeOK* is an invariant—that is, it is true in every state of every behavior allowed by the specification.

$$\begin{aligned}
&TypeOK \triangleq \\
&\quad \land rnd \in [Acceptor \rightarrow Nat] \\
&\quad \land vrnd \in [Acceptor \rightarrow Nat] \\
&\quad \land vval \in [Acceptor \rightarrow Val \cup \{any\}]
\end{aligned}$$

$\land$ *crnd* $\in [\mathit{Coord} \rightarrow \mathit{Nat}]$
$\land$ *cval* $\in [\mathit{Coord} \rightarrow \mathit{Val} \cup \{\mathit{any}, \mathit{none}\}]$
$\land$ *amLeader* $\in [\mathit{Coord} \rightarrow \text{BOOLEAN}]$
$\land$ *sentMsg* $\in \text{SUBSET } \mathit{Message}$
$\land$ *proposed* $\in \text{SUBSET } \mathit{Val}$
$\land$ *learned* $\in \text{SUBSET } \mathit{Val}$
$\land$ *goodSet* $\subseteq \mathit{Acceptor} \cup \mathit{Coord}$

Init is the initial predicate that describes the initial values of all the variables.

```
$\mathit{Init} \triangleq$
  $\land \mathit{rnd} = [a \in \mathit{Acceptor} \mapsto 0]$
  $\land \mathit{vrnd} = [a \in \mathit{Acceptor} \mapsto 0]$
  $\land \mathit{vval} = [a \in \mathit{Acceptor} \mapsto \mathit{any}]$
  $\land \mathit{crnd} = [c \in \mathit{Coord} \mapsto 0]$
  $\land \mathit{cval} = [c \in \mathit{Coord} \mapsto \mathit{none}]$
  $\land \mathit{amLeader} \in [\mathit{Coord} \rightarrow \text{BOOLEAN}]$
  $\land \mathit{sentMsg} = \{\}$
  $\land \mathit{proposed} = \{\}$
  $\land \mathit{learned} = \{\}$
  $\land \mathit{goodSet} \in \text{SUBSET } (\mathit{Acceptor} \cup \mathit{Coord})$
```

Action Definitions

Send(m) describes the state change that represents the sending of message m. It is used as a conjunct in defining the algorithm actions.

$\mathit{Send}(m) \triangleq \mathit{sentMsg}' = \mathit{sentMsg} \cup \{m\}$

Coordinator Actions

Action *Phase*1*a*(c, i) specifies the execution of phase 1a of round i by coordinator c, described in Section 2.2.1 (on page 5) and refined by CA2$'$ (Section 3.3, page 22).

```
$\mathit{Phase1a}(c, i) \triangleq$
  $\land \mathit{amLeader}[c]$
  $\land c = \mathit{CoordOf}(i)$
  $\land \mathit{crnd}[c] < i$
  $\land \lor \mathit{crnd}[c] = 0$
    $\lor \exists m \in \mathit{sentMsg} : \land \mathit{crnd}[c] < m.\mathit{rnd}$
                          $\land m.\mathit{rnd} < i$
    $\lor \land \mathit{crnd}[c] \in \mathit{FastNum}$
      $\land i \in \mathit{ClassicNum}$
    ∧ $crnd' = [crnd$ EXCEPT $![c] = i]$
    ∧ $cval' = [cval$ EXCEPT $![c] = none]$
    ∧ $Send([type \mapsto$ "phase1a", $rnd \mapsto i])$
    ∧ UNCHANGED $\langle aVars, oVars \rangle$
```

MsgsFrom$(Q, i, phase)$ is defined to be the set of messages in *sentMsg* of type *phase* (which may equal "phase1b" or "phase2b") sent in round i by the acceptors in the set Q.

```
$MsgsFrom(Q, i, phase) \triangleq$
    $\{m \in sentMsg : (m.type = phase) \land (m.acc \in Q) \land (m.rnd = i)\}$
```

If M is the set of round i phase 1b messages sent by the acceptors in a quorum Q, then *IsPickableVal*(Q, i, M, v) is true iff the rule of Figure 2 (page 20) allows the coordinator to send the value v in a phase 2a message for round i.

```
$IsPickableVal(Q, i, M, v) \triangleq$
  LET $vr(a) \triangleq$ (CHOOSE $m \in M : m.acc = a).vrnd$
      $vv(a) \triangleq$ (CHOOSE $m \in M : m.acc = a).vval$
      $k \triangleq Max(\{vr(a) : a \in Q\})$
      $V \triangleq \{vv(a) : a \in \{b \in Q : vr(b) = k\}\}$
      $O4(w) \triangleq \exists R \in Quorum(k) :$
                    $\forall a \in R \cap Q : (vr(a) = k) \land (vv(a) = w)$
  IN   IF $k = 0$ THEN ∨ $v \in proposed$
                       ∨ ∧ $i \in FastNum$
                         ∧ $v = any$
              ELSE IF $Cardinality(V) = 1$
                     THEN $v \in V$
                     ELSE IF $\exists w \in V : O4(w)$
                            THEN $v =$ CHOOSE $w \in V : O4(w)$
                            ELSE $v \in proposed$
```

Action *Phase*2*a*(c, v) specifies the execution of phase 2a by coordinator c with value v, as described in Section 2.2.1 (on page 5) and Section 2.2.2 (page 6), and refined by CA2′ (Section 3.3, page 22).

```
$Phase2a(c, v) \triangleq$
  LET $i \triangleq crnd[c]$
  IN   ∧ $i \neq 0$
       ∧ $cval[c] = none$
       ∧ $amLeader[c]$
       ∧ $\exists Q \in Quorum(i) :$
           ∧ $\forall a \in Q : \exists m \in MsgsFrom(Q, i,$ "phase1b"$) : m.acc = a$
           ∧ $IsPickableVal(Q, i, MsgsFrom(Q, i,$ "phase1b"$), v)$
       ∧ $cval' = [cval$ EXCEPT $![c] = v]$
```

```
        $\land\ Send([type \mapsto \text{"phase2a"},\ rnd \mapsto i,\ val \mapsto v])$
        $\land$ UNCHANGED $\langle crnd,\ aVars,\ oVars\rangle$
```

$P2bToP1b(Q, i)$ is defined to be the set of round $i+1$ phase 1b messages implied by the round i phase 2b messages sent by the acceptors in the set Q, as explained in Section 3.2.

```
$P2bToP1b(Q,\ i) \triangleq$
  $\{[type \mapsto \text{"phase1b"},\ rnd \mapsto i+1,\ vrnd \mapsto i,$
      $vval \mapsto m.val,\ acc \mapsto m.acc] : m \in MsgsFrom(Q,\ i,\ \text{"phase2b"})\}$
```

Action $CoordinatedRecovery(c, v)$ specifies the coordinated recovery described in Section 3.2, page 20. With this action, coordinator c attempts to recover from a collision in round $crnd[c]$ by sending round $crnd[c]+1$ phase 2a messages for the value v. Although CA2$'$ (Section 3.3, page 22) implies that this action should be performed only if $crnd[c]+1$ is a classic round, that restriction is not required for correctness and is omitted from the specification.

```
$CoordinatedRecovery(c,\ v) \triangleq$
  LET $i \triangleq crnd[c]$
  IN  $\land\ amLeader[c]$
      $\land\ cval[c] = any$
      $\land\ c = CoordOf(i+1)$
      $\land\ \exists\, Q \in Quorum(i+1):$
          $\land\ \forall\, a \in Q : \exists\, m \in P2bToP1b(Q,\ i) : m.acc = a$
          $\land\ IsPickableVal(Q,\ i+1,\ P2bToP1b(Q,\ i),\ v)$
      $\land\ cval' = [cval$ EXCEPT $![c] = v]$
      $\land\ crnd' = [crnd$ EXCEPT $![c] = i+1]$
      $\land\ Send([type \mapsto \text{"phase2a"},\ rnd \mapsto i+1,\ val \mapsto v])$
      $\land$ UNCHANGED $\langle aVars,\ oVars\rangle$
```

$coordLastMsg(c)$ is defined to be the last message that coordinator c sent, if $crnd[c] > 0$.

```
$coordLastMsg(c) \triangleq$
  IF $cval[c] = none$
    THEN $[type \mapsto \text{"phase1a"},\ rnd \mapsto crnd[c]]$
    ELSE $[type \mapsto \text{"phase2a"},\ rnd \mapsto crnd[c],\ val \mapsto cval[c]]$
```

In action $CoordRetransmit(c)$, coordinator c retransmits the last message it sent. This action is a stuttering action (meaning it does not change the value of any variable, so it is a no-op) if that message is still in $sentMsg$. However, this action is needed because c might have failed after first sending the message and subsequently have been repaired after the message was removed from $sentMsg$.

```
$CoordRetransmit(c) \triangleq$
  $\land\ amLeader[c]$
  $\land\ crnd[c] \neq 0$
```

```
    ∧ $Send(coordLastMsg(c))$
    ∧ UNCHANGED $\langle aVars, cVars, oVars \rangle$
```

CoordNext(c) is the next-state action of coordinator c—that is, the disjunct of the algorithm's complete next-state action that represents actions of that coordinator.

```
$CoordNext(c) \triangleq$
    ∨ $\exists\, i \in RNum : Phase1a(c, i)$
    ∨ $\exists\, v \in Val \cup \{any\} : Phase2a(c, v)$
    ∨ $\exists\, v \in Val : CoordinatedRecovery(c, v)$
    ∨ $CoordRetransmit(c)$
```

Acceptor Actions

Action *Phase*1*b*(i, a) specifies the execution of phase 1b for round i by acceptor a, described in Section 2.2.1 on page 5.

```
$Phase1b(i, a) \triangleq$
    ∧ $rnd[a] < i$
    ∧ $[type \mapsto \text{"phase1a"}, rnd \mapsto i] \in sentMsg$
    ∧ $rnd' = [rnd$ EXCEPT $![a] = i]$
    ∧ $Send([type \mapsto \text{"phase1b"}, rnd \mapsto i, vrnd \mapsto vrnd[a], vval \mapsto vval[a],$
                $acc \mapsto a])$
    ∧ UNCHANGED $\langle cVars, oVars, vrnd, vval \rangle$
```

Action *Phase*2*b*(i, a, v) specifies the execution of phase 2b for round i by acceptor a, upon receipt of either a phase 2a message or a proposal (for a fast round) with value v. It is described in Section 2.2.1 on page 5 and Section 3.1 on page 17.

```
$Phase2b(i, a, v) \triangleq$
    ∧ $rnd[a] \leq i$
    ∧ $vrnd[a] < i$
    ∧ $\exists\, m \in sentMsg :$
          ∧ $m.type = \text{"phase2a"}$
          ∧ $m.rnd = i$
          ∧ ∨ $m.val = v$
            ∨ ∧ $m.val = any$
              ∧ $v \in proposed$
    ∧ $rnd'  = [rnd$  EXCEPT $![a] = i]$
    ∧ $vrnd' = [vrnd$ EXCEPT $![a] = i]$
    ∧ $vval' = [vval$ EXCEPT $![a] = v]$
    ∧ $Send([type \mapsto \text{"phase2b"}, rnd \mapsto i, val \mapsto v, acc \mapsto a])$
    ∧ UNCHANGED $\langle cVars, oVars \rangle$
```

Action $UncoordinatedRecovery(i, a, v)$ specifies uncoordinated recovery, described in Section 3.2 on page 21. With this action, acceptor a attempts to recover from a collision in round i by sending a round $i+1$ phase 2b message with value v.

```
$UncoordinatedRecovery(i, a, v) \triangleq$
  $\land\ i + 1 \in FastNum$
  $\land\ rnd[a] \leq i$
  $\land\ \exists\, Q \in Quorum(i + 1) :$
        $\land\ \forall\, b \in Q : \exists\, m \in P2bToP1b(Q, i) : m.acc = b$
        $\land\ IsPickableVal(Q, i + 1, P2bToP1b(Q, i), v)$
  $\land\ rnd' \ \ = [rnd \ \ \text{EXCEPT } ![a] = i + 1]$
  $\land\ vrnd' = [vrnd \text{ EXCEPT } ![a] = i + 1]$
  $\land\ vval' \ = [vval \text{ EXCEPT } ![a] \ = v]$
  $\land\ Send([type \mapsto \text{“phase2b”}, rnd \mapsto i + 1, val \mapsto v, acc \mapsto a])$
  $\land\ \text{UNCHANGED } \langle cVars, oVars \rangle$
```

$accLastMsg(a)$ is defined to be the last message sent by acceptor a, if $rnd[a] > 0$.

```
$accLastMsg(a) \triangleq$
  IF $vrnd[a] < rnd[a]$
    THEN $[type \mapsto \text{“phase1b”}, rnd \mapsto rnd[a], vrnd \mapsto vrnd[a],$
              $vval \mapsto vval[a], acc \mapsto a]$
    ELSE  $[type \mapsto \text{“phase2b”}, rnd \mapsto rnd[a], val \mapsto vval[a],$
              $acc \mapsto a]$
```

In action $AcceptorRetransmit(a)$, acceptor a retransmits the last message it sent.

```
$AcceptorRetransmit(a) \triangleq$
  $\land\ rnd[a] \neq 0$
  $\land\ Send(accLastMsg(a))$
  $\land\ \text{UNCHANGED } \langle aVars, cVars, oVars \rangle$
```

$AcceptorNext(a)$ is the next-state action of acceptor a—that is, the disjunct of the next-state action that represents actions of that acceptor.

```
$AcceptorNext(a) \triangleq$
  $\lor\ \exists\, i \in RNum : \lor\ Phase1b(i, a)$
                        $\lor\ \exists\, v \in Val : Phase2b(i, a, v)$
  $\lor\ \exists\, i \in FastNum, v \in Val : UncoordinatedRecovery(i, a, v)$
  $\lor\ AcceptorRetransmit(a)$
```

Other Actions

Action $Propose(v)$ represents the proposal of a value v by some proposer.

```
$Propose(v) \triangleq$
  $\land\ proposed' = proposed \cup \{v\}$
  $\land\ \text{UNCHANGED}\ \langle aVars, cVars, amLeader, sentMsg, learned, goodSet \rangle$
```

Action *Learn*(v) represents the learning of a value v by some learner.

```
$Learn(v) \triangleq$
  $\land\ \exists\, i \in RNum :$
        $\exists\, Q \in Quorum(i) :$
          $\forall\, a \in Q :$
            $\exists\, m \in sentMsg :\ \land\ m.type = \text{"phase2b"}$
                                 $\land\ m.rnd = i$
                                 $\land\ m.val = v$
                                 $\land\ m.acc = a$
  $\land\ learned' = learned \cup \{v\}$
  $\land\ \text{UNCHANGED}$
        $\langle aVars, cVars, amLeader, sentMsg, proposed, goodSet \rangle$
```

Action *LeaderSelection* allows an arbitrary change to the values of *amLeader*[c], for all coordinators c. Since this action may be performed at any time, the specification makes no assumption about the outcome of leader selection. (However, progress is guaranteed only under an assumption about the values of *amLeader*[c].)

```
$LeaderSelection \triangleq$
  $\land\ amLeader' \in [Coord \to \text{BOOLEAN}]$
  $\land\ \text{UNCHANGED}\ \langle aVars, cVars, sentMsg, proposed, learned, goodSet \rangle$
```

Action *FailOrRepair* allows an arbitrary change to the set *goodSet*. Since this action may be performed at any time, the specification makes no assumption about what agents are good. (However, progress is guaranteed only under an assumption about the value of *goodSet*.)

```
$FailOrRepair \triangleq$
  $\land\ goodSet' \in \text{SUBSET}\ (Coord \cup Acceptor)$
  $\land\ \text{UNCHANGED}\ \langle aVars, cVars, amLeader, sentMsg, proposed, learned \rangle$
```

Action *LoseMsg*(m) removes message m from *sentMsg*. It is always enabled unless m is the last message sent by an acceptor or coordinator in *goodSet*. Hence, the only assumption the specification makes about message loss is that the last message sent by an agent in *goodSet* is not lost. Because *sentMsg* includes messages in an agent's output buffer, this effectively means that a non-failed process always has the last message it sent in its output buffer, ready to be retransmitted.

```
$LoseMsg(m) \triangleq$
  $\land\ \neg \lor\ \land\ m.type \in \{\text{"phase1a"}, \text{"phase2a"}\}$
           $\land\ m = coordLastMsg(CoordOf(m.rnd))$
```

```
        $\land\ \mathit{CoordOf}(m.rnd) \in \mathit{goodSet}$
        $\land\ \mathit{amLeader}[\mathit{CoordOf}(m.rnd)]$
      $\lor\ \land\ m.type \in \{\text{“phase1b”}, \text{“phase2b”}\}$
        $\land\ m = \mathit{accLastMsg}(m.acc)$
        $\land\ m.acc \in \mathit{goodSet}$
  $\land\ \mathit{sentMsg}' = \mathit{sentMsg} \setminus \{m\}$
  $\land\ \text{UNCHANGED}\ \langle \mathit{aVars}, \mathit{cVars}, \mathit{oVars} \rangle$
```

Action *OtherAction* is the disjunction of all actions other than ones performed by acceptors or coordinators, plus the *LeaderSelection* action (which represents leader-selection actions performed by the coordinators).

```
$\mathit{OtherAction} \triangleq$
  $\lor\ \exists\, v \in \mathit{Val} : \mathit{Propose}(v) \lor \mathit{Learn}(v)$
  $\lor\ \mathit{LeaderSelection} \lor \mathit{FailOrRepair}$
  $\lor\ \exists\, m \in \mathit{sentMsg} : \mathit{LoseMsg}(m)$
```

Next is the algorithm's complete next-state action.

```
$\mathit{Next} \triangleq$
  $\lor\ \exists\, c \in \mathit{Coord} : \mathit{CoordNext}(c)$
  $\lor\ \exists\, a \in \mathit{Acceptor} : \mathit{AcceptorNext}(a)$
  $\lor\ \mathit{OtherAction}$
```

Temporal Formulas

Formula *Fairness* specifies the fairness requirements as the conjunction of weak fairnes formulas. Intuitively, it states approximately the following:

> A coordinator c in *goodSet* must perform some action if it can, and it must perform a $\mathit{Phase1a}(c, i)$ action for a classic round i if it can.
>
> An acceptor in *goodSet* must perform some action if it can.
>
> A value that can be learned must be learned.

It is not obvious that these fairness requirements suffice to imply the progress property, and that fairness of each individual acceptor and coordinator action is not needed. Part of the reason is that formula *Fairness* does not allow an agent in *goodSet* to do nothing but *Retransmit* actions if another of its actions is enabled, since all but the first retransmission would be a stuttering step, and weak fairness of an action A requires a non-stuttering A step to occur if it is enabled.

```
$\mathit{Fairness} \triangleq$
  $\land\ \forall\, c \in \mathit{Coord} :$
      $\land\ \mathrm{WF}_{vars}((c \in \mathit{goodSet}) \land \mathit{CoordNext}(c))$
      $\land\ \mathrm{WF}_{vars}((c \in \mathit{goodSet}) \land (\exists\, i \in \mathit{ClassicNum} : \mathit{Phase1a}(c, i)))$
  $\land\ \forall\, a \in \mathit{Acceptor} : \mathrm{WF}_{vars}((a \in \mathit{goodSet}) \land \mathit{AcceptorNext}(a))$
  $\land\ \forall\, v \in \mathit{Val} : \mathrm{WF}_{vars}(\mathit{Learn}(v))$
```

Formula *Spec* is the complete specification of the Fast Paxos algorithm.

$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$

Nontriviality asserts that every learned value has been proposed, and *Consistency* asserts that at most one value has been learned. The Nontriviality and Consistency conditions for consensus (Section 2.1) are equivalent to the invariance of these state predicates.

$Nontriviality \triangleq learned \subseteq proposed$
$Consistency \triangleq Cardinality(learned) \leq 1$

The following theorem asserts that the state predicates *TypeOK*, *Nontriviality*, and *Consistency* are invariants of specification *Spec*, which implies that *Spec* satisfies the safety properties of a consensus algorithm. It was checked by the TLC model checker on models that were too small to find a real bug in the algorithm but would have detected most simple errors in the specification.

THEOREM $Spec \Rightarrow \Box(TypeOK \land Nontriviality \land Consistency)$

Because the specification does not explicitly mention proposers and learners, condition $LA(p, l, c, Q)$ described on page 10 of Section 2.3.1 is replaced by $LA(c, Q)$, which depends only on c and Q. Instead of asserting that some particular proposer p has proposed a value, it asserts that some value has been proposed.

```
$LA(c, Q) \triangleq$
    $\land \{c\} \cup Q \subseteq goodSet$
    $\land proposed \neq \{\}$
    $\land \forall ll \in Coord : amLeader[ll] \equiv (c = ll)$
```

The following theorem asserts that *Spec* satisfies the progress property of Fast Paxos, described in Sections 2.3 and 3.3. The temporal formula $\Diamond\Box LA(c, Q)$ asserts that $LA(c, Q)$ holds from some time on, and $\Diamond(learned \neq \{\})$ asserts that some value is eventually learned.

```
THEOREM $\land Spec$
        $\land \exists Q \in$ SUBSET $Acceptor :$
              $\land \forall i \in ClassicNum : Q \in Quorum(i)$
              $\land \exists c \in Coord : \Diamond\Box LA(c, Q)$
        $\Rightarrow \Diamond(learned \neq \{\})$
```