On Collision-fast Atomic Broadcast

Rodrigo Schmidt
EPFL, USI

Lásaro Camargos
Unicamp, USI

Fernando Pedone
USI

January, 2007

Abstract

Atomic Broadcast, an important abstraction in dependable distributed computing, is usually implemented by many instances of the well-known consensus problem. Some asynchronous consensus algorithms achieve the optimal latency of two (message) steps but cannot guarantee this latency even in good runs, with quick message delivery and no crashes. This is due to *collisions*, a result of concurrent proposals. Collision-fast consensus algorithms, which decide within two steps in good runs, exist under certain conditions. Their direct application to solving atomic broadcast, though, does not guarantee delivery in two steps for all messages unless a single failure is tolerated. We show a simple way to build a fault-tolerant collision-fast Atomic Broadcast algorithm based on a variation of the consensus problem we call M-Consensus. Our solution to M-Consensus extends the Paxos protocol to allow multiple processes, instead of the single leader, to have their proposals learned in two steps.

Contents

1 Introduction

Atomic broadcast is a primitive that allows components of a distributed system to agree on an ever-growing sequence of broadcast messages [4]. The problem can be defined in terms of *proposers* and *learners*, which respectively broadcast and deliver messages [6, 8]. To implement a replicated state machine [5], for example, clients playing the role of proposers issue commands to the replicas through atomic broadcast. The replicas learn the agreed sequence of commands and execute them in order. Assuming commands are deterministic, all replicas undergo the same state transitions.

Atomic broadcast is often solved using the consensus problem as a building block. In fact, the two problems are equivalent with respect to solvability but consensus has a simpler definition since learners must eventually learn only a single value out of the set of proposed ones. The equivalence between consensus and atomic broadcast, though, brings out some interesting results. First, it extends to atomic broadcast the famous FLP impossibility result stating that consensus is not deterministically solvable in asynchronous systems subject to failures [3]. Moreover, since the reduction from consensus to atomic broadcast is direct (learners learn only the first element of the agreed sequence), any lower bounds for consensus also apply to atomic broadcast.

Generally speaking, one can solve atomic broadcast by means of a totally ordered succession of consensus instances. A sender that wants to broadcast a message proposes it in the first instance for which the sender has not proposed or learned anything yet. Consensus ensures that the decision reliably reaches all nonfaulty learners, and the delivery order is given by the ordering of the instances, that is, the i^{th} instance's decision gives the i^{th} element in the learned sequence. Proposers must be also consensus learners so that they can check if their proposal in some instance was decided or not and repropose it in a different instance in case it was not the consensus decision. Clearly, the performance of any implementation of this general approach is highly dependent on the consensus protocol it hinges upon.

This solution to atomic broadcast has a problem, though: Because the decision of each instance is bounded to a single proposal, messages proposed but not decided in a given instance must be reproposed on subsequent instances until they get decided, increasing their delivery delay. Notice that even implementations in which proposals are composed of sets of messages (e.g., [2]) may suffer from this problem since there is no guarantee that all processes propose always the same sets in all instances.

Some consensus algorithms for the asynchronous model rely on a leader to coordinate the agreement procedure and this can be used to bypass the

problem above. In such algorithms, proposals are sent to the leader, which selects one as the possible decision and continues with the algorithm execution. In an atomic broadcast implementation, all the consensus instances could share the same leader, as done in Paxos [6]. Instead of selecting an instance of consensus by the time a message is broadcast, proposals could be just forwarded to the leader. The leader selects the first instance it has not used and continues with the algorithm as if the received proposal related to that instance. This gives to the atomic broadcast implementation the same latency as the consensus protocol—three message steps in general, or two for messages broadcast by the leader.

There are consensus protocols that can achieve the latency of two message steps for multiple proposers by employing stricter conditions for a proposal to be decided (e.g, Fast Paxos [9]). In such algorithms, there is no leader involved in the general case for getting a proposal decided but quorums are necessarily bigger. Moreover, the absence of a leader to circumvent the FLP impossibilty result creates a problem called *collision*, which happens when two concurrent proposals are issued but none gets decided after two message steps [10]. To solve a collision, extra message steps are required. It is possible to ensure a latency of two message steps in normal runs and avoid collisions. An asynchronous consensus algorithm that achieves this is called collision-fast. In [10], Lamport states the two conditions in which collision-fast asynchronous consensus algorithms are possible. The first case restricts fault tolerance to a single failure and is solved by a simple variant of Paxos, which allows the optimization we mentioned in the previous paragraph for atomic broadcast. As for the second condition, which does not restrict the number of failures, its algorithm applied to solving atomic broadcast cannot solve the problem of different proposals for the same instance of consensus resulting in a single decision. Thus, non-decided proposals must be resubmitted in different instances, delaying their learning.

Indeed, when more than one process can fail, it seems impossible to use the standard reduction from atomic broadcast to consensus and obtain a collision-fast atomic broadcast protocol, that is, an atomic broadcast implementation in which messages broadcast are delivered within two message steps in normal runs. Differently, we reduce atomic broadcast to a variation of consensus we call M-Consensus. In M-Consensus, processes decide not on a single proposed value, but on a bounded composition of them. This way, if concurrent proposals happen, *all* of them may take part in the final decision. To implement atomic broadcast, we use a succession of M-Consensus instances as done before with standard consensus. Wise collision-fast implementations of M-Consensus, however, can produce a collision-fast

atomic broadcast. Collision-fast Paxos, our solution to M-Consensus, extends the original Paxos algorithm to allow multiple proposers, and not only the leader, to have their proposals decided in two message steps. As we show in the paper, our protocol can be used to implement a collision-fast atomic broadcast that tolerates as many failures as the original Paxos.

2 Model and Definitions

2.1 Model

Instead of using processes, we state our definitions in terms of agents that perform actions in the system; processes can aggregate the roles of several agents. We assume an asynchronous crash-recovery model in which agents communicate by exchanging messages with no bounds on the time it takes for messages to be transmitted or actions to be executed. Messages can be lost or duplicated but not corrupted; agents can fail by stopping only and never perform incorrect actions. Agents are assumed to have some sort of local stable storage to keep their state in between failures so that finite periods of absence are not distinguishable from excessive slowness. Although we assume agents may recover, they are not obliged to do so once they have failed. For simplicity, an agent is considered to be nonfaulty iff it never fails.

2.2 Atomic Broadcast

Given two sets of agents, namely *proposers* and *learners*, the atomic broadcast problem consists of ensuring that messages broadcast by proposers are eventually delivered by all learners, in the same order. As in [8], we phrase the problem as the agreement on an ever-growing sequence of broadcast messages, of which learners learn increasing prefixes. First, though, we introduce the required notation. We represent a *sequence* s as the tuple of its elements $\langle v_1, v_2, \ldots, v_n \rangle$, where n is the length of s and v_i equals $s[i]$, the sequence's i^{th} element. We say that sequence s is a prefix of sequence t, noted as $s \sqsubseteq t$, iff the length of s is less than or equal to the length of t and, for all i from 1 to the length of s, $s[i] = t[i]$; s and t are equal iff $s \sqsubseteq t$ and $t \sqsubseteq s$. The empty sequence $\langle\rangle$ has length zero and is a prefix of any other sequence. Atomic broadcast's safety properties can then be defined as follows, where $delivered[l]$ refers to the sequence of messages delivered by learner l, initially $\langle\rangle$.

Nontriviality For any learner l, $delivered[l]$ contains only broadcast messages and no duplicates.

Stability For any learner l, if $delivered[l] = s$ at some time, then $s \sqsubseteq delivered[l]$ at all later times.

Consistency For any pair of learners $l1$ and $l2$, either $delivered[l1] \sqsubseteq delivered[l2]$ or $delivered[l2] \sqsubseteq delivered[l1]$.

In an actual system, client applications broadcast commands and learn the result of their execution, tasks possibly associated with proposers and learners in our model. Since we cannot require clients not to fail, we define liveness in terms of another set of agents: the *acceptors*. Let a *quorum* be any finite set of acceptors large enough to ensure liveness. The liveness property of atomic broadcast is defined as follows.

Liveness For any proposer p and learner l, if p, l and a quorum of acceptors are nonfaulty and p broadcasts a message m, then eventually $delivered[l]$ contains m.

2.3 Algorithms

In this section, we formally define atomic broadcast algorithms and what it means for them to be collision-fast. The formal definitions we give are mostly borrowed from [10]; as in that work, we start by describing events.

An *event* is an action performed at some agent either spontaneously or triggered by the reception of a message. Each event e performed by agent e_{agent} sends exactly one message e_{msg}, receivable by any agent, including itself. We assume that events are totally ordered at the agents performing them, that is, we assume that each event e performed by agent e_{agent} is uniquely identified by the positive integer e_{num}, indicating that e was the ${e_{num}}^{\text{th}}$ event performed by e_{agent}. For an event e triggered by the reception of a message, we let e_{rcvd} equal the triple $\langle m, a, i \rangle$, where m is the received message, a is the agent that sent it, and i the index e_{num} of m's sending event e.

A *scenario* is the set of events performed in some single (partial) execution of an algorithm. For every event in a scenario, all other events that could have causally influenced it must also be in the scenario. To formally define a scenario, we let $\preceq_S$ be, for any set S of events, the transitive closure of the relation $\rightarrow$ on S such that $e \rightarrow f$ iff either (i) $e_{agent} = f_{agent}$ and $e_{num} \leq f_{num}$ or (ii) f is a message-receiving event and $f_{rcvd} = \langle e_{msg}, e_{agent}, e_{num} \rangle$.

Definition 1 (Scenario [10]) *A* scenario S *is a set of events such that:*

- *for any agent a, the set of events in S performed by a consists of k_a events numbered from 1 through k_a, for some natural number k_a;*
- *for every message-receiving event $e \in S$, there exists $d \in S$, $d \neq e$, such that $e_{rcvd} = \langle d_{msg}, d_{agent}, d_{num} \rangle$; and*
- *$\preceq_S$ is a partial order on S.*

A scenario obtained by removing the last events of a scenario S, according to the precedence relation $\preceq_S$, is called a *prefix* of S.[1]

Definition 2 (Prefix [10]) *A subset S of a scenario T is a* prefix *of T, written $S \sqsubseteq T$, iff for any events d in T and e in S, if $d \preceq_T e$ then d is in S.*

An algorithm can be seen as the set of non-empty scenarios it allows. However, we are only interested in algorithms that are compliant with our model. We define an *asynchronous algorithm* as follows, where $Agents(S)$ is the set of agents that performed events in S.

Definition 3 (Asynchronous Algorithm [10]) *An* asynchronous algorithm *Alg is a set of scenarios such that:*

- *every prefix of a scenario in Alg is in Alg; and*
- *if T and U are scenarios of Alg and S is a prefix of both T and U such that $Agents(T \setminus S)$ and $Agents(U \setminus S)$ are disjoint sets, then $T \cup U$ is a scenario of Alg.*

We define a *source* of a scenario S as an event $e \in S$ that is minimal in the ordering $\preceq_S$, and we let the depth of an event be the number of message steps that precede the event.

Definition 4 (Event Depth [10]) *The* depth *of an event e in a scenario S equals 0 if e is a source of S, otherwise it equals the maximum of*

(i) the depths of all events d with $d_{agent} = e_{agent}$ and $d_{num} < e_{num}$, and

(ii) if e is an event that receives a message sent by event b, then 1 plus the depth of b.

[1] For simplicity, we use the same nomenclature and notation to define the prefix relation between sequences, scenarios, and, as shown later, v-mappings. These sets are different and used in different contexts, which makes us believe this cannot be a source of confusion.

We now must define what a collision-fast atomic broadcast protocol is. Due to the space limitations, though, the definition we present considers only scenarios in which messages are broadcast in the source events. As a result, an algorithm might be collision-fast according to this simplified definition even if it does not ensure the same delivery latency for non-source broadcasts. Nonetheless, we believe that algorithms that satify our definition can be usually adapted to ensure the same delivery latency for all messages broadcast in normal runs, as this is the case for our solution.

A *normal scenario* is one in which the execution starts by one or more agents atomically broadcasting, messages are not lost or duplicated, timeouts do not occur, messages are received in FIFO order, and no event receives a message with depth lower than its own minus one, that is, message reception is not delayed for two message steps or more.

Definition 5 (Normal Scenario [10]) *A scenario S is* normal *iff:*

- *the only sources of S are (atomic) broadcast events;*
- *the message sent by any single event is not received twice by the same agent;*
- *every non-source event is a message receiving event;*
- *if $d1$ and $d2$ are events in S with $d1_{agent} = d2_{agent}$ and $d1 \preceq_S d2$, and $e2$ is an event in S that receives the message sent by $d2$, then there exists an event $e1$ in S with $e1_{agent} = e2_{agent}$ and $e1 \preceq_S e2$ such that $e1$ receives the message sent by $d1$; and,*
- *if d and e are events in S and e receives the messages sent by d, then e_{depth} equals 1 plus d_{depth} in S.*

Our definition of collision-fast atomic broadcast states that the messages initially broadcast are delivered in two message steps. In order to measure that, we use the definition below.

Definition 6 (Complete to Depth [10]) *An agent a is* complete to depth δ *in a scenario S iff either $\delta = 0$ or every agent in $Agents(S)$ is complete to depth $\delta - 1$ and a receives every message sent by an event in S with depth less than δ.*

We consider an atomic broadcast algorithm to be collision-fast iff there is a set M of agents and a set P of at least two proposers such that all messages

initially broadcast by any subset O of the proposers in P are delivered by a learner l when l is complete to depth 2 in a normal scenario in which no agent in $M \cup O \cup \{l\}$ crashes. Our formal definition below is derived from the definition of Collision-fast Accepting in [10].

Definition 7 (Collision-fast Algorithm) *An asynchronous atomic broadcast algorithm Alg is* collision-fast *iff there is a set M of agents and a set P of proposers with at least two proposers such that, for every nonempty subset $\{p_1, \ldots, p_k\}$ of P with p_i all distinct:*

- *for any broadcastable messages $m_1, \ldots, m_k$ there is a scenario $\{e_1, \ldots, e_k\}$ in Alg such that each e_i is a source event in which p_i broadcasts m_i; and,*
- *for every learner l and every normal scenario S of Alg with $Agents(S) = \{l, p_1, \ldots, p_k\} \cup M$ that contains $\{e_1, \ldots, e_k\}$ as a prefix, if l is complete to depth 2 in S, then $delivered[l]$ contains $m_1, \ldots, m_k$.*

3 M-Consensus

In the M-Consensus problem, where M stands for mapping, agents must agree on an increasing mapping from proposers to either proposed values or to the special value *Nil*. Before formalizing the problem, though, we define the value mapping data structure, v-mapping for short, it depends upon.

3.1 Value Mapping Sets

In order to introduce v-mappings, we must define some function notation. As usual, we let $f(d)$ be the result of function f for its domain element d. We represent the set of all functions with domain D and range R by $[D \to R]$, and the domain of a function f by $Dom(f)$. Moreover, we assume the existence of a special function $\perp$ such that $Dom(\perp) = \{\}$.

A value mapping set is a data structure defined in terms of sets *Domain* and *Value*. Each pair $\langle Domain, Value \rangle$ corresponds to a different set *ValMap* of value mappings, defined as all functions from subsets of *Domain* to $Value \cup \{Nil\}$, where *Nil* is a special value not present in *Value*. More formally, $ValMap = \bigcup\{[D \to R] : D \subseteq Domain \wedge R = Value \cup \{Nil\}\}$. A v-mapping is therefore a function that maps some elements of *Domain* to either a value in *Value* or *Nil*. Notice that, since $\{\} \subseteq Domain$ for any set *Domain*, $\perp$ is present in every v-mapping set. To ease the presentation, hereinafter we

consistently use uppercase letters for values in $Value \cup \{Nil\}$ and lowercase letters for v-mappings in *ValMap*.

We call a pair $\langle d, V \rangle$, where $d \in Domain$ and $V \in Value \cup \{Nil\}$, a *single mapping*, or s-mapping for short, and define the append operation $v \bullet \langle d, V \rangle$, where v is a v-mapping and $\langle d, v \rangle$ is an s-mapping, to equal v-mapping f such that (i) $Dom(f) = Dom(v) \cup \{d\}$ and (ii) $\forall q \in Dom(f)$: IF $q \in Dom(v)$ THEN $f(q) = v(q)$ ELSE $f(q) = V$. Informally, $v \bullet \langle d, V \rangle$ extends v with the s-mapping $\langle d, V \rangle$ iff d is not in the domain of v. The append operator defines a partial order relation on a v-mapping set. We say that v-mapping v is a *prefix* of v-mapping w, and w is an extension of v ($v \sqsubseteq w$), iff w can be generated from v by a series of append operations. The precedence between v and w can be easily checked since $v \sqsubseteq w$ iff $Dom(v) \subseteq Dom(w)$ and $\forall d \in Dom(v) : v(d) = w(d)$. We define $v \sqsubset w$ to be true iff $v \sqsubseteq w$ and $v \neq w$.

Given a set $T \subseteq ValMap$, we say that v-mapping v is a lower bound of T iff $v \sqsubseteq w$ for all w in T. A greatest lower bound (glb) of T is a lower bound v of T such that $w \sqsubseteq v$ for every lower bound w of T, and we represent it by $\sqcap T$. Similarly, we say that v is an upper bound of T iff $w \sqsubseteq v$ for all w in T. A least upper bound (lub) of T is an upper bound v of T such that $v \sqsubseteq w$ for every upper bound w of T, and we represent it by $\sqcup T$. For simplicity of notation, we use $v \sqcap w$ and $v \sqcup w$ to represent $\sqcap\{v, w\}$ and $\sqcup\{v, w\}$, respectively. There is always a unique glb for a set T of v-mappings. The existence of a lub, however, depends on whether the set T is compatible, but if it exists, then it is unique. Two v-mappings v and w are defined to be *compatible* iff there exists a v-mapping u such that $v \sqsubseteq u$ and $w \sqsubseteq u$. A set S of v-mappings is compatible iff its elements are pairwise compatible. Compatibility can be easily checked since two v-mappings are compatible iff the elements in the intersection of their domains are mapped to the same values.

We say that a value mapping is *complete* iff its domain equals *Domain*. It is easy to see that a complete v-mapping does not have any strict extension, since no append operation applied to it can result in a different v-mapping. An interesting complete v-mapping is the one that maps every element in *Domain* to *Nil*. This v-mapping is independent of the set *Value* and, for this reason, we call it the *trivial* v-mapping. A v-mapping is *nontrivial* iff it is different from the trivial one.

3.2 Problem definition

As we have done for atomic broadcast, we define the M-Consensus problem in terms of the sets of proposer, acceptor, and learner agents, and a set of proposable values. The problem considers the v-mapping set with *Domain* equal to the set of proposers and *Value* equal to the set of proposable values. Proposers propose values and learners learn v-mappings that can differ but must always be compatible, can only be extended, and must eventually equal the same complete nontrivial v-mapping. We say that a v-mapping is *proposed* iff all elements of its domain are mapped either to *Nil* or to a proposed value and we let $learned[l]$ represent the v-mapping currently learned by learner l, initially $\perp$. Based on that, the properties of M-Consensus are defined as follows:

Nontriviality For any learner l, $learned[l]$ is always a nontrivial proposed v-mapping.

Stability For any learner l, if $learned[l] = v$ at some time, then $v \sqsubseteq learned[l]$ at all later times.

Consistency The set of learned v-mappings is always compatible and has a nontrivial lub.

Liveness For any proposer p and learner l, if p, l and a quorum of acceptors are nonfaulty and p proposes a value, then eventually $learned[l]$ is complete.

Learners initially know $\perp$, which is a valid prefix for any v-mapping. As proposers make proposals, learners can extend their learned v-mappings as long as they are always proposed and nontrivial. Note that a v-mapping that maps all its domain to *Nil* but does not cover all elements in *Domain* is nontrivial and can be learned by a learner, which is not a problem since the remaining elements of *Domain* can still be mapped to some value. Consistency ensures that all currently learned values can be extended to a common v-mapping that satisfies the Nontriviality property. The existence of a nontrivial lub is also implied by Liveness and Nontriviality, but its presence in the Consistency property makes the problem specification machine-closed [1]. The Liveness property states that all correct learners will eventually learn a complete v-mapping, which implies that, like consensus and differently from atomic broadcast, an instance of M-Consensus eventually terminates.

With respect to solvability, M-Consensus is equivalent to consensus. It is easy to see that an algorithm that solves consensus can solve M-Consensus

by just having learners learn a mapping in which a specific proposer is mapped to the decided value and all the others are mapped to *Nil*. An algorithm that solves M-Consensus also trivially solves consensus by just totally ordering the set of proposers and picking up the value mapped to the first proposer not mapped to *Nil*. Actually, this equivalence grants to M-Consensus all known lower bounds for consensus. The advantage of M-Consensus, though, has to do with the implementation of atomic broadcast since it allows two concurrent proposals to appear in the problem solution, mapped to different proposers. This avoids the proposal collision problem existent in consensus-based atomic broadcast and explained in Section 1.

4 Collision-Fast Paxos

This section describes our solution to M-Consensus, which we later employ to solve atomic broadcast. An unambiguous TLA$^+$ [7] specification of the algoritm and its proofs of correctness can be found in the appendix.

4.1 Basic Algorithm

We first describe our basic algorithm, which satisfies safety but does not guarantee liveness, a topic addressed in the next section. The algorithm is structured in rounds and the only assumption we make about them is that they are totally ordered by a relation $\leq$. For simplicity, it can be assumed that rounds correspond to the natural numbers unless we explicitly state it differently (Section 4.2). As in the original Paxos protocol [6], every round has a single coordinator assigned to it. Coordinators represent a different sort of agent besides proposers, acceptors, and learners.

We also assign to each round r a subset of the proposers we call the collision-fast proposers of r. The collision-fast proposers of a round are the only proposers allowed to have their proposals learned in two communication steps at that round. As we explain later, making all proposers collision-fast for all rounds would restrict the algorithm's resilience.

At some round r, a collision-fast proposer p *fast-proposes* an s-mapping $\langle p, V\rangle$ at most once. It does that when it has a value to be proposed or when it notices that another collision-fast proposer of round r has fast proposed a non-*Nil* value—a situation in which p fast-proposes $\langle p, Nil\rangle$. If the fast proposal contains a mapping with a proposed value, it is sent to the acceptors and other collision-fast proposers; otherwise it is sent directly to the learners. An acceptor may *accept* multiple v-mappings as long as the newly accepted v-mapping extends the previous one. The v-mappings

accepted by the acceptors are generated from the non-*Nil* s-mappings fast-proposed and, therefore, always map at least one proposer to a non-*Nil* value.

We say that a v-mapping v is *chosen* at round r iff there exists a (possibly empty) subset P of the collision-fast proposers of r such that the two conditions below hold:

- every proposer $p \in P$ has fast-proposed s-mapping $\langle p, Nil \rangle$ and
- there exists a quorum Q of acceptors such that every acceptor $a \in Q$ has accepted a v-mapping w such that v is a prefix of w extended with $\langle p, Nil \rangle$ for every proposer $p \in P$.

More intuitively, one can think that if a collision-fast proposer p has fast-proposed $\langle p, Nil \rangle$ at round r, then every acceptor that has accepted or later accepts some v-mapping at r will "automatically", though not explicitly, extend it with $\langle p, Nil \rangle$. Thinking this way, a v-mapping is chosen at round r if it is a prefix of every v-mapping accepted by some quorum acceptor Q at r.

Chosen v-mappings are guaranteed to be compatible and a learner can extend $learned[l]$ by setting it to the lub between $learned[l]$ and any chosen v-mapping. If at least one collision-fast proposer fast-proposes a value, no process crashes, and messages are correctly delivered, it is easy to see that learners learn a complete nontrivial v-mapping within two message steps. However, new rounds might have to be started due to failures. To ensure consistency in this case v-mappings chosen in some round must be made compatible with v-mappings chosen in other rounds.

The algorithm keeps the invariant that if a v-mapping is or might be chosen at some round r then any v-mapping accepted at a higher-numbered round extends the possibly chosen one. This is guaranteed by the actions taken to start a new round. A new round's coordinator queries a quorum of acceptors to discover if some v-mapping has been or might be chosen at a lower-numbered round. If this is the case, the coordinator extends such v-mapping with *Nil* mappings to make it complete and sends it to the acceptors for it to be accepted and chosen directly. If no v-mapping has been or might be chosen at a lower-numbered round, the collision-fast proposers of the current round are notified that they can fast-propose for that round (collision-fast proposers wait for this confirmation before fast-proposing at some round).

For the coordinator to be able to identify if some value has been or might be chosen at a lower-numbered round by just querying a quorum of

acceptors, we need the following assumption about quorums:

Assumption 1 (Quorum Requirement) *If Q and R are quorums, then $Q \cap R \neq \emptyset$.*

In fact, any general algorithm for asynchronous consensus (and, therefore, M-Consensus) must satisfy a similar requirement, as shown by the Accepting Lemma in [10]. A simple way to ensure this is defining quorums as any majority of the acceptors.

To make our algorithm description precise, we must explain the variables required by each process. A proposer p has the following variables:

$prnd[p]$ The current round of p. Initially 0.

$pval[p]$ The value p has fast-proposed at round $prnd[p]$ or special value *none* if p has not fast-proposed anything at round $prnd[p]$. Initially *none*.

The variables of a coordinator c are:

$crnd[c]$ The current round of c. Initially 0.

$cval[c]$ The initial v-mapping for round $crnd[c]$, if c has already queried a quorum of acceptors for $crnd[c]$ or special value *none* otherwise. Initially $\perp$ for the coordinator of round 0 and *none* for all the others.

An acceptor a keeps three variables:

$rnd[a]$ The current round of a. Initially 0.

$vrnd[a]$ The round at which a has accepted its latest value. Initially 0.

$vval[a]$ The v-mapping a has accepted at $vrnd[a]$ if it has accepted something at $vrnd[a]$, or special value *none* otherwise. Initially *none*.

Each learner l keeps only the v-mapping it has learned so far.

$learned[l]$ The v-mapping currently learned by l. Initially $\perp$.

In the following, we present the basic atomic actions that compose the algorithm.

$Propose(p, V)$ Executed by proposer p when it wants to propose value V. p sends message $\langle$"`propose`", $V\rangle$ to some collision-fast proposer for round $prnd[p]$. It is just a local message if p is a collision-fast proposer of $prnd[p]$.

*Phase*1$a(c, r)$ Executed by coordinator c to start round r. It is enabled iff:

- c is the coordinator of round r and
- $crnd[c] < r$.

It sets $crnd[c]$ to r, $cval[c]$ to *none*, and sends message $\langle$"`1a`", $r\rangle$ to the acceptors.

*Phase*1$b(a, r)$ Executed by acceptor a, for round r. It is enabled iff:

- a has received a $\langle$"`1a`", $r\rangle$ message and
- $rnd[a] < r$

It sets $rnd[a]$ to r and sends message $\langle$"`1b`", $r, a, vrnd[a], vval[a]\rangle$ to the coordinator of round r. Setting $rnd[a]$ to r makes sure that no mapping will be further accepted by a at a round lower than r and the "`1b`" message tells the coordinator of r that the last value accepted by a for a round lower than r was $vval[a]$ at round $vrnd[a]$.

*Phase*2*Start*(c, r) Executed by coordinator c of round r. This action picks up an initial v-mapping for round r based on the "`1b`" messages the coordinator c received for round r from a quorum of acceptors. It is enabled iff:

- $r = crnd[c]$,
- $cval[c] = none$, and
- c has received a "`1b`" message for round r from every acceptor in a quorum Q.

Let k be the highest *vrnd* field received in the "`1b`" messages mentioned above and let S be the set of all v-mappings (different from *none*) received in the "`1b`" messages with field *vrnd* equal to k. If S is empty, then no v-mapping has been or might be chosen at a lower-numbered round and c can pick up v-mapping $\perp$ to start round r. In this case, it sets $cval[c]$ to $\perp$ and sends message $\langle$"`2S`", $r, \perp\rangle$ to all proposers, allowing them to fast-propose when they are ready. Acceptors need not be notified in this case.

If S is not empty, then it might be the case that some v-mapping has been or might be chosen at a round lower than or equal to k. As mentioned before, the algorithm guarantees that if a v-mapping was or might be chosen at some round lower than k, then it is a prefix of all

values accepted in k, including those in S. Moreover, if any v-mapping has been or might be chosen at round k, then, by the quorum assumption, it must have been accepted by some acceptor in Q and, thus, is present in S. As we explain in action $Phase2b(a, r)$, v-mappings accepted by acceptors for the same round are always compatible and this obviously guarantees the compatibility of set S. Therefore, $\sqcup S$ extends both the v-mappings possibly chosen at rounds lower than k and the v-mappings possibly chosen at k. Because acceptors only accept v-mappings that map at least one proposer to a non-*Nil* value, $\sqcup S$ also satisfies this property and extending it with s-mappings $\langle p, Nil\rangle$ for every proposer p does not generate the trivial mapping. Let v be $\sqcup S$ extended with $\langle p, Nil\rangle$ for every proposer p; coordinator c sets $cval[c]$ to v and sends message $\langle$"`2S`"$, r, v\rangle$ to all acceptors and proposers.

$Phase2Prepare(p, r)$ Executed by proposer p, for round r. It is enabled iff:

- $prnd[p] < r$ and
- p has received a message $\langle$"`2S`"$, r, v\rangle$.

First, it sets $prnd[p]$ to r. If $v = \bot$, it sets $pval[p]$ to *none*; otherwise, it sets $pval[p]$ to $v(p)$. Recall, from action $Phase2Start(c, r)$ above, that a "`2S`" message for any round contains either $\bot$ or a complete v-mapping.

$Phase2a(p, r, V)$ Executed by proposer p, where r is the current round of p and V is either a proposed value or *Nil*. It is enabled iff:

- $prnd[p] = r$,
- p is a collision-fast proposer of r,
- $pval[p] = none$, and
- either p has received message $\langle$"`propose`"$, V\rangle$ or V equals *Nil* and p has received message $\langle$"`2a`"$, r, \langle q, W\rangle\rangle$, where $\langle q, W\rangle$ is an s-mapping from any proposer q to a non-*Nil* value W.

It sets $pval[p]$ to V and sends message $\langle$"`2a`"$, r, \langle p, V\rangle\rangle$ either to the acceptors and other collision-fast proposers of r, if V does not equal *Nil*, or directly to the learners otherwise. In this action, proposer p fast-proposes, giving its opinion about the value it should be mapped to. It is triggered by the receipt of a "`propose`" message with a proposed value (a local 0-latency message if p sent it to itself) or by the receipt of a "`2a`" message from another collision-fast proposer, which forces p to set its opinion to *Nil*.

*Phase*2*b*(a, r) Executed by acceptor a, for round r and v-mapping v. It is enabled iff:

- $rnd[a] \leq r$ and
- Either one of the two following conditions is satisfied:
  a) a has received message $\langle$"2S"$, r, v\rangle$, where $v \neq \perp$, and $vrnd[a] < r$ or $vval[a] = none$
  b) a has received message $\langle$"2a"$, r, \langle p, V\rangle\rangle$, where $V \neq Nil$.

It sets $rnd[a]$ and $vrnd[a]$ to r and changes $vval[a]$ depending on whether condition (a) or (b) above is satisfied. If condition (a) is true, it sets $vval[a]$ to v. If condition (b) is true and $vrnd[a] < r$ or $vval[a] = none$, then it sets $vval[a]$ to $Bottom \bullet \langle p, V\rangle$ extended with $\langle q, Nil\rangle$ for every proposer q that is not collision-fast for r; otherwise, it sets $vval[a]$ to its previous value extended with $\langle p, V\rangle$, that is, $vval[a] \bullet \langle p, V\rangle$. It then sends message $\langle$"2b"$, r, a, vval[a]\rangle$ to all learners, with the updated value of $vval[a]$.

Condition (a) implies that the coordinator of round r has picked up v-mapping $v \neq \perp$ for round r based on the votes of a quorum of acceptors for lower-numbered rounds. As explained in action *Phase*2*Start*(c, r), this v-mapping v is complete and different from the trivial mapping.

Condition (b) implies that the coordinator of round r has picked up $\perp$ for the initial v-mapping of r and collision-fast proposers were allowed to fast-propose. In this case, the first mapping acceptor a accepts for round r maps the proposer p that sent the "2a" message to the (non-*Nil*) value it sent and maps every proposer that is not collision-fast for r to *Nil*, since they are not allowed to fast-propose. When a receives the "2a" messages from other collision-fast proposers of round r with non-*Nil* values, a just appends the received s-mapping to the previously accepted v-mapping.

It is not possible that, for some round r, an acceptor executes this action due to condition (a) and another acceptor executes it due to condition (b). If acceptors execute this action for a round r satisfying condition (a), they must accept the same complete v-mapping v. If acceptors execute this action for r satisfying condition (b), they must accept v-mappings that map a proposer p either to *Nil*, if p is not collision-fast for r, or to the value p sent in its "2a" message, if p is collision-fast for r. Since no proposer can send different "2a" messages for the same round, all v-mappings accepted by condition (b) must be

compatible. This arguments shows that it is safe to calculate the lub of any set of v-mappings accepted for the same round, as done in action $Phase2Start(c, r)$.

Learn(l) Executed by learner l. It is enabled iff l has received "`2b`" messages for some round r from a quorum Q and message $\langle$"`2a`"$, r, \langle p, Nil\rangle\rangle$ from every proposer p in a (possibly empty) subset P of the collision fast proposers of round r. It calculates the lub of the chosen v-mappings based on the received information in order to update the currently learned v-mapping of l. Let *Q2bVals* be the set of all v-mappings received in the "`2b`" messages for round i from acceptors in Q, and let *newv* be $\sqcap$*Q2bVals* extended with $\langle p, Nil\rangle$ for every proposer p in P. The action sets $learned[l]$ to $learned[l] \sqcup newv$.

4.2 Ensuring Liveness

The previous actions ensure safety, but if messages are lost, coordinators or collision-fast proposers crash, or coordinators keep on starting new rounds, then they will not ensure progress. We now extend the algorithm for that. Some of the assumptions we make are very basic. It is clear that no algorithm can ensure progress if messages can be indiscriminately lost and non-crashed agents indefinitely refuse to take actions that are enabled. Therefore, we assume that if agents a and b do not crash and a keeps resending message m to b, then b eventually receives m. Moreover, we assume weak fairness on the actions an agent may take, that is, no action remains enabled forever without being executed. We tacitly assume that an action is enabled only if its agent is not crashed.

The FLP result and the equivalence between consensus and M-Consensus with respect to solvability imply that these assumptions are not enough to ensure liveness for M-Consensus. As in the original Paxos protocol, we circumvent FLP by eventually electing a distinguished coordinator—the leader—responsible for starting new rounds. For it to work, we require also that every coordinator be reponsible for infinitely many higher-numbered rounds, which is easily ensured by having round numbers defined as tuples $\langle n, c\rangle$ where n is a natural number and c is its coordinator identifier.

When the leader starts a round and picks up $\bot$ as its initial value, the round will only succeed in getting a complete v-mapping chosen and learned if all its collision-fast proposers remain up. This is inherent to collision-fast consensus algorithms like ours as implies the Collision-fast Learning Theorem of [10] and, in fact, it is the main reason why we designed Collision-

fast Paxos so that the set of collision-fast proposers depends on the round; had we done it differently, the failure of any collision-fast proposer would not allow our algorithm to become collision-fast again. As a result, the leader must be able to somehow identify when a collision-fast proposer of the current round has crashed in order to start a new one. We assume that a coordinator c that believes itself to be the leader keeps a set $activep[c]$ with all the proposers it believes to be currently up. We assume this set can take any valid value but, in order to ensure liveness, it must eventually satisfy some conditions we show later in this section.

For progress, we need to make a number of small changes to the algorithm we presented in Section 4.1:

- We add "c believes itself to be the leader" as a pre-condition to actions $Phase1a(c, r)$ and $Phase2Start(c, r)$.
- If an acceptor a receives a "`1a`", "`2S`", or "`2a`" message for round r such that $r < rnd[a]$ and the coordinators of r and $rnd[a]$ differ, then a sends a special message to the coordinator of r to inform that round $rnd[a]$ was initiated.
- The same sort of special message is sent if a proposer p receives a "`2S`" message for round r such that $r < prnd[p]$ and the coordinators of r and $prnd[p]$ differ.
- Besides the first modification, coordinator c executes action $Phase1a(c, r)$ only if either it receives a special message informing of round j ($r > j > crnd[c]$) was initiated, or the set of collision fast-proposers of $crnd[c]$ is not a subset of $activep[c]$ but the set of collision-fast proposers of r is.
- Each proposer p that has sent a "`propose`" message keeps resending it to one of the collision-fast proposers of $prnd[p]$.
- Each coordinator that believes itself to be the leader keeps resending, to all its original receivers, the last "`1a`" or "`2S`" message it sent.
- Each proposer p that has executed action $Phase2a(p, r, V)$, for round $r = prnd[p]$ and any V, keeps resending the last "`2a`" message it sent.
- Each acceptor keeps resending the last "`1b`" or "`2b`" message it sent.

These changes do not affect safety because they incur new actions that do not change the algorithm's variables and make some actions' pre-conditions more restrictive only.

Except for the conditions related to new variable *activep*, the liveness assumption of Collision-fast Paxos is the same as the one of the original protocol (c.f. Section 2.3 of [9]). We define $LA(p, l, c, Q)$ for any proposer p, learner l, coordinator c, and quorum Q of acceptors, to be the conjunction of the following conditions:

- $\{p, l, c\} \cup Q$ are not crashed.
- p has proposed a value.
- c is the only coordinator that believes itself to be the leader.
- All proposers in $activep[c]$ are not crashed.
- For every round $r > crnd[c]$, c is the coordinator of a round $s > r$ whose collision-fast proposers are all in $activep[c]$.
- $activep[c]$ is a subset of all its future values.

If $LA(p, l, c, Q)$ holds for some proposer p, coordinator c, and quorum Q, from some point in time on, then eventually l learns a complete v-mapping. If every coordinator is itself the only collision-fast proposer for infinitely higher-numbered rounds that it coordinates, then Collision-fast Paxos could ensure liveness in the same situations where Paxos would. In fact, a round in which the only collision-fast proposer is the round coordinator itself implements a standard Paxos round.

As we mentioned before, the set of collision-fast proposers is defined per round, so that failed proposers can be excluded from the set to allow collision-fast termination even after failures. For that, we extend round numbers with the round's set of collision-fast proposers, defining round numbers as tuples of the form $\langle n, c, cf \rangle$, where n is a natural number, c is the round's coordinator, and cf is the sorted list of the round's collision-fast proposers. It is clear from this definition that a lexicographical comparison induces a total order on the round numbers. To ensure the uniqueness of the special round *Zero*, it is defined *a priori* as $\langle 0, c, cf \rangle$, for some coordinator c and list cf. This scheme grants to each coordinator an infinite number of rounds for every possible set of collision-fast proposers.

5 Atomic Broadcast

Solving atomic broadcast with M-Consensus is simple; achieving a collision-fast solution, though, depends on the M-Consensus algorithm in use. We

first present the general approach and then extend it to use Collision-fast Paxos. A complete TLA$^+$ specification of our atomic broadcast algorithm is given in the appendix.

5.1 General Approach

To implement atomic broadcast, we use infinitely many M-Consensus instances, each one uniquely identified by a natural number. To differentiate messages and variables of different instances we superscript them with the instance's identification (e.g., $learned^i$, "$\texttt{1b}^j$"). Atomic broadcast proposers act both as proposers and learners in each of the M-Consensus instances.

To broadcast a message m, a proposer p proposes m in the smallest instance of M-Consensus i in which it has neither proposed nor learned anything yet. Being also a learner, p eventually learns the decision of i, and checks if there exists some proposer q such that $learned^i[p](q) = m$. If there is, then p knows that m was successfully broadcast and will eventually be delivered by all nonfaulty learners; otherwise, p re-proposes m in the next free M-Consensus instance. This procedure can be executed in parallel for many messages.

Assuming there is a known total order of proposers, learner l builds sequence $delivered[l]$ by considering each M-Consensus instance in order and then, for each proposer p, also in order, checking if p has something mapped to it on that instance. If so, l appends the mapped value to $delivered[l]$ iff it is different from Nil and not yet contained by $delivered[l]$. If p has nothing mapped to, l stops the procedure because the current instance is not complete yet.

5.2 Collision-fast Paxos Approach

Using Collision-fast Paxos, all M-Consensus instances can share the same coordinator. This also allows us to keep all instances synchronized with respect to their current round in all agents. As a result, variables $rnd[a]$, $prnd[p]$, and $crnd[c]$ can be shared amongst all instances. The other variables are not shared but could be allocated for an instance only when their value changes from the initial one. When a coordinator executes action $Phase1a(c, r)$, it does that for all instances and sends a single "$\texttt{1a}$" message. An acceptors a that executes action $Phase1b(a, r)$, also does that for all instances and aggregates all "$\texttt{1b}^i$" messages it should send in a single one. Only a finite number of instances will have $vval^i[a] \neq none$, which allows the compression of this message to a finite size. After collecting these com-

posite "1b" messages from a quorum of acceptors, a coordinator c executes $Phase2Start(c, r)$ for all instances and, similarly, generates a composite "2S" message containing the "$2S^i$" message of every instance i. A proposer p that receives such composite "2S" message, simply executes $Phase2Prepare(p, r)$ for all instances.

The actions above are executed only when the leader changes. During normal execution, things are simpler. A collision-fast proposer that wants to atomically broadcast a message M fast-proposes M in the first instance i for which $pval^i[p] = none$ by executing action $Phase2a^i(p, r, M)$. If everything goes fine, the message will be eventually learned and delivered; if failures or suspicions prevent the normal case, eventually $pval^i[p]$ will change from M to Nil due to a "2S" message and p will notice that it will have to repropose M in another instance. When a proposer that is not collision-fast for its current round wants to atomically broadcast a message, it simply forwards it to one of the collision-fast proposers. Notice that, since Collision-fast Paxos ensures that a collision-fast proposer eventually knows if its fast proposal was learned or not, this implementation does not require that proposers be learners too. Acceptors and Learners execute actions $Phase2b(a, r)$ and $Learn(l)$ independently for each instance. As for progress, this atomic broadcast implementation has the same liveness condition as Collision-fast Paxos.

As discussed in Section 4.1, in the normal case, if a collision-fast proposer p fast-proposes a message, then a v-mapping containing it is learned in two message steps. If there are no concurrent (non-Nil) fast-proposals for the same instance, this v-mapping will be complete. Otherwise, a learner complete to depth 2 plus the depth of p's fast proposal will learn a complete v-mapping containing all fast proposals, since all are learned in two steps. Because a message to be broadcast by a collision-fast proposer never waits to be fast-proposed in some instance and a collision-fast proposer leaves no gaps between instances, this atomic broadcast algorithm is collision-fast. In fact, according to our definition it is collision-fast for P equal to the collision-fast proposers of round 0 and M equal to $Q \cup P$ where Q is a quorum.

6 Conclusion

We have discussed the implementation of a collision-fast atomic broadcast protocol. Since the traditional approach to implement atomic broadcast based on standard consensus cannot result on a resilient collision-fast implementation, we have proposed a new agreement problem called M-Consensus that allows multiple proposals to take part in the final decision. Our so-

lution to M-Consensus, called Collision-fast Paxos, is an extension of the Paxos protocol in which a number of proposers can have their proposals as part of the final decision in two message steps. Using Collision-fast Paxos to implement a collision-fast atomic broadcast algorithm is simple and provides a very efficient fault-tolerant protocol.

Although there exist atomic broadcast algorithms that can deliver messages within two steps in some optimistic runs (e.g., [11, 12, 13]), the only protocol we found in the literature that is truly collision-fast is [14]. It tolerates more than a single failure but, instead of relying on consensus, it extends the timestamp-based algorithm presented by Lamport in [5], which supports our claim that it is impossible to implement a collision-fast atomic broadcast algorithm that tolerates more than a single failure based on standard consensus. In contrast to the approach in [14], ours considers a weaker model, where processes can crash and recover, and messages can be lost or duplicated. Moreover, our algorithm allows reconfiguration in case collision-fast proposers fail so that execution can become collision-fast again, which is not the case for [14] when failures happen.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

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

[4] V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition, 1993.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004.

[9] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[10] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[11] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[12] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[13] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proc. of the 21th IEEE Symp. on Reliable Distributed Systems (SRDS'02)*, pages 92–101, Osaka, Japan, Oct. 2002.

[14] P. Zielinski. Low-latency atomic broadcast in the presence of contention. In *Proc. of the* 20^{th} *Intl. Symposium on Distributed Computing, DISC'2006*, pages 505–519, 2006.

A The Safety of Collision-fast Paxos

This section presents the proof that Collision-fast Paxos satisfies the safety properties of M-Consensus.

A.1 Preliminaries

We start by defining a special data structure we call a *ballot array*. Our definition is highly-inspired by the data structure with the same name presented in [8]. A ballot array represents the voting history of a set of acceptors, that is, the history of v-mappings accepted by acceptors on different rounds. For every acceptor a, it keeps the current round of a, $\widehat{bA}_a$, and, for every acceptor a and round r, the vote a has cast at r, $bA_a[r]$. If an acceptor has not cast a vote at round r, then $bA_a[r]$ equals special value *none*. To ease the design of our algorithms, we force acceptors to vote only for v-mappings that have at least one element of their domain mapped to a non-*Nil* value. A v-mapping that satisfies this constraint is called a *valued* v-mapping. The complete definition of a ballot array is given below.

Definition 8 (Ballot Array) *A* ballot array bA *is a mapping that assigns to each acceptor a a round $\widehat{bA}_a$ and to each acceptor a and round r a value $bA_a[r]$ that is a v-mapping or equals none, such that for every acceptor a:*

- *The set of rounds m with $bA_a[m] \neq none$ is finite,*
- *$bA_a[r] = none$ for all rounds $r > \widehat{bA}_a$, and*
- *$bA_a[r]$ is either none or a valued v-mapping for all rounds r.*

As mentioned in Section 4.1, a v-mapping is learned depending not only on the votes cast by acceptors but also on the *Nil* values proposed by proposers. Because of that, we define another data structure we call a *proposal array*. A proposal array represents a history of proposals made by proposers at different rounds. It keeps, for every proposer p and round r, the (possibly *Nil*) value p has proposed at round r, or special value *none* if p has not proposed at round r.

Definition 9 (Proposal Array) *A* proposal array pA *is a mapping that assigns to each proposer p and round r a value $pA_p[r]$ that is either a proposable value, special value Nil, or special value none.*

In fact, proposal arrays are important only for their *Nil* proposals because these proposals are used to define a *chosen* v-mapping, that is, a v-mapping that can be safely learned by a learner without jeopardizing consistency. Before we give a formal definition for a chosen v-mapping, though, we have to introduce the operator $NilExtension(v, P)$, which we refer to as the *Nil*-extension of v for P where v is a v-mapping and P is a set of proposers. This operator returns *none* if v equals *none*; otherwise, it is defined as v-mapping w satisfying the three conditions below:

1. $Dom(w) = Dom(v) \cup P$
2. $\forall p \in Dom(w) \cap Dom(v) : w(p) = v(p)$
3. $\forall p \in Dom(w) \setminus Dom(v) : w(p) = Nil$

Intuitively, $NilExtension(v, P)$ extends v by mapping each proposer in $P \setminus Dom(v)$ to *Nil*. A different but equivalent definition of this operator appears in TLA$^+$ module *PaxosConstants* in Section D.

We say that a v-mapping v is *chosen at* some round r in pair $\langle bA, pA \rangle$, where bA is a ballot array and pA is a proposal array, iff there is a quorum Q of acceptors and a (possibly empty) set P of collision-fast proposers for r that have proposed *Nil* at round r such that, for every acceptor a in Q, v is a prefix of the *NilExtension* of the v-mapping a has accepted at round r for P. For completeness, we define that *none* is not a prefix or an extension of any v-mapping.

Definition 10 (Chosen at) *A v-mapping v is* chosen at *round r in $\langle bA, pA \rangle$, where bA is a ballot array and pA is a proposal array, iff there exists a set P of collision-fast proposers for r and a quorum Q such that:*

- $\forall p \in P : pA_p[r] = Nil$
- $\forall q \in Q : v \sqsubseteq NilExtension(bA_a[r], P)$

A v-mapping v is chosen in $\langle bA, pA \rangle$ *iff it is chosen at some round r in $\langle bA, pA \rangle$.*

We say that a v-mapping v is *choosable at* some round r if it is possible to extend the voting history represented by bA and the proposal array pA so that v satisfies the condition above to be considered *chosen* at r in $\langle bA, pA \rangle$.

Definition 11 (Choosable at) *A v-mapping v is* choosable at *round r in pair $\langle bA, pA \rangle$, where bA is a ballot array and pA is a proposal array, if, and*

only if, considering P to be the set of proposers p such that $pA_p[r]$ is either Nil or none, there exists a quorum Q such that $v \sqsubseteq NilExtension(bA_a[r], P)$ for every acceptor a in Q with $\widehat{bA}_a > r$.

We say that a v-mapping is safe at some round in a pair $\langle bA, pA\rangle$, where bA is a ballot array and pA is a proposal array, if it extends all v-mappings that are choosable at lower-numbered rounds in $\langle bA, pA\rangle$. We also say that a a pair $\langle bA, pA\rangle$ is safe if all v-mappings that acceptors have voted for in bA are *safe* at the rounds they were accepted in $\langle bA, pA\rangle$.

Definition 12 (Safe at) *A v-mapping v is* safe at *round r in $\langle bA, pA\rangle$, where bA is a ballot array and pA is a proposal array, iff $w \sqsubseteq v$ for every round $k < r$ and every v-mapping w that is choosable at k. A pair $\langle bA, pA\rangle$ is* safe *iff for every acceptor a and balnum k, if $bA_a[k] \neq none$ then it is safe at k in $\langle bA, pA\rangle$.*

The proposition below states that if a pair of ballot and proposal arrays is safe, then all its chosen v-mappings are compatible.

Proposition 1 *Let bA be a ballot array and pA be a proposal array, if $\langle bA, pA\rangle$ is safe, then the set of values that are chosen in $\langle bA, pA\rangle$ is compatible.*

PROOF: By the definition of Consistency, it suffices to
ASSUME: 1. $\langle bA, pA\rangle$ is safe
2. v-mapping v is chosen at round r in $\langle bA, pA\rangle$
3. v-mapping w is chosen at round $s \geq r$ in $\langle bA, pA\rangle$
PROVE: v and w are compatible.

1. Choose a quorum Q_v and set P_v of collision-fast proposers for r such that
   - $\forall p \in P_v : pA_p[r] = Nil$
   - $\forall q \in Q_v : v \sqsubseteq NilExtension(bA_a[r], P_v)$

   PROOF: This follows from proof assumption 2 and the definition of *chosen at*.
2. Choose a quorum Q_w and set P_w of collision-fast proposers for s such that
   - $\forall p \in P_w : pA_p[r] = Nil$
   - $\forall q \in Q_w : v \sqsubseteq NilExtension(bA_a[r], P_w)$

   PROOF: This follows from proof assumption 3 and the definition of *chosen at*.

3. Case: $r = s$
   3.1. Choose an acceptor a in $Q_v \cap Q_w$
   Proof: a exists by the Quorum Requirement (Assumption 1).
   3.2. $v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
      3.2.1. $v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v)$
      Proof: By steps 1 and 3.1.
      3.2.2. $\mathit{NilExtension}(bA_a[r], P_v) \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
      Proof: By the definition of *Nil*-extension.
      3.2.3. Q.E.D.
   3.3. $w \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
      3.3.1. $w \sqsubseteq \mathit{NilExtension}(bA_a[r], P_w)$
      Proof: By steps 2 and 3.1.
      3.3.2. $\mathit{NilExtension}(bA_a[r], P_w) \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
      Proof: By the definition of *Nil*-extension.
      3.3.3. Q.E.D.
   3.4. Q.E.D.
   Proof: By steps 3.2 and 3.3 and the definition of compatible.
4. Case: $r < s$
   4.1. v is choosable at r in $\langle bA, pA \rangle$
   Proof: By the definition of choosable, any v-mapping chosen at some round is also choosable at it.
   4.2. Choose any acceptor a in Q_w
   Proof: The Quorum Requirement implies that quorums cannot be empty.
   4.3. $v \sqsubseteq bA_a[s]$
   Proof: By the fact that $\langle bA, pA \rangle$ is safe (proof assumption 1).
   4.4. $w \sqsubseteq \mathit{NilExtension}(bA_a[s], P_w)$
   Proof: By Step 2.
   4.5. Q.E.D.
   Proof: Step 4.3 and the definition of *Nil*-extension imply that $v \sqsubseteq \mathit{NilExtension}(bA_a[s], P_w)$. Step 4.4, and the definition of compatible complete the proof.
5. Q.E.D.
   Proof: All cases were considered since $r \leq s$ according to proof assumption 3.

We define a pair $\langle bA, pA \rangle$ to be *conservative* iff all v-mappings accepted by any acceptors a and b at the same round are compatible and if the v-mapping accepted by b maps some proposer not mapped by the v-mapping accepted by a, then this proposer is mapped to the value it has proposed

for that round.

Definition 13 (Conservative) *A pair* $\langle bA, pA\rangle$ *is* conservative *iff for every round* r *and all acceptors* a *and* b*, if* $bA_a[r]$ *and* $bA_b[r]$ *are different from none, then the two conditions below hold:*

- $bA_a[r]$ *and* $bA_b[r]$ *are compatible and*
- $\forall p \in Dom(b) \setminus Dom(a) : bA_b[r][p] = pA_p[r]$.

Below we present the definition of operator $ProvedSafe(Q, r, bA)$, which returns a v-mapping that is proved to be safe at round r in $\langle bA, pA\rangle$ for any proposal array pA based only on the votes of acceptors in Q, given that $\langle bA, pA\rangle$ is safe and conservative, and, for every acceptor a in Q, $\widehat{bA}_a \geq r$. In the definition below, we let *Proposer* be the set of all proposers.

Definition 14 (ProvedSafe) *For any round* r*, quorum* Q*, and ballot array* bA*, let:*

- $KS \triangleq \{i \in RNum \mid (i < r) \wedge (\exists a \in Q : bA_a[i] \neq none])\}$
- $k \triangleq Max(KS)$
- $AS \triangleq \{a \in Q : bA_a[k] \neq none\}$
- $G \triangleq \{bA_a[k] : a \in S\}$

If $KS = \{\}$*, then* $ProvedSafe(Q, r, bA)$ *is defined to equal* $\perp$*; otherwise,* $ProvedSafe(Q, r, bA)$ *is defined to equal* $NilExtension(\sqcup G, Proposer)$*, where Proposer is the set of all proposers.*

The proposition below states that the value returned by $ProvedSafe(Q, r, bA)$ is indeed safe at r in $\langle bA, pA\rangle$ if $\langle bA, pA\rangle$ is safe and conservative and, for every acceptor a in Q, $\widehat{bA}_a \geq r$.

Proposition 2 *For any round* r*, quorum* Q*, ballot array* bA*, and proposal array* pA*, if*

- $\langle bA, pA\rangle$ *is safe,*
- $\langle bA, pA\rangle$ *is conservative, and*
- $\widehat{bA}_a \geq r$ *for all* $a \in Q$*,*

then $ProvedSafe(Q, r, bA)$ *is safe at* r *in* $\langle bA, pA\rangle$.

ASSUME: There exist round r, quorum Q, ballot array bA, and proposal array pA such that:
1. $\langle bA, pA \rangle$ is safe
2. $\langle bA, pA \rangle$ is conservative
3. $\forall a \in Q : \widehat{bA}_a \geq r$

PROVE: *ProvedSafe*(Q, r, bA) is a v-mapping safe at r in $\langle bA, pA \rangle$

LET: KS be the set KS in the definition of *ProvedSafe* for Q,r, and bA.

1. CASE: KS is empty
   1.1. No v-mapping v is choosable at a round $s < r$ in $\langle bA, pA \rangle$
   PROOF: By the definition of *choosable at*, it suffices to
   LET: P be the set $\{p \in Proposer : pA_p[s] = Nil\}$
   ASSUME: There exist v-mapping v, round $s < r$, and quorum Q_v such that $v \sqsubseteq NilExtension(bA_a[s], P)$, for every acceptor a in Q_v with $\widehat{bA}_a > s$
   PROVE: FALSE
   1.1.1. Choose any acceptor $a \in Q_v \cap Q$
   PROOF: Such acceptor exists because of the Quorum Requirement (Assumption 1).
   1.1.2. $\widehat{bA}_a > s$
   PROOF: Since a belongs to Q, proof assumption 3 states that $\widehat{bA}_a \geq r$, and the assumption of step 1.1 states that $r > s$. As a result, $\widehat{bA}_a > s$.
   1.1.3. $v \not\sqsubseteq NilExtension(bA_a[s], P)$
   PROOF: By the definition of KS, if KS is empty, then $bA_a[s]$ must equal *none*, otherwise KS would have s as an element. By definition, any *Nil*-extension of *none* equals *none* and no v-mapping is a prefix of *none*.
   1.1.4. Q.E.D.
   PROOF: Steps 1.1.2 and 1.1.3 and the fact that a belongs to Q_v given by step 1.1.1 contradict the assumption of step 1.1.
   1.2. Q.E.D.
   PROOF: By step 1.1, any v-mapping is safe at r in bA. Therefore, $\perp$, which is the value returned by *ProvedSafe*(Q, r, bA), is safe too.
2. CASE: KS is not empty
   2.1. Choose round k and sets AS and G so that they satisfy the definitions of k, AS and G in the definition of *ProvedSafe* for Q,r, and bA.
   PROOF: k exists since KS is not empty. AS and G exist because k exists.
   2.2. AS and G are not empty.

PROOF: Given that k belongs to KS, there is at least one acceptor a in Q such that $bA_a[k] \neq none$.

2.3. G is compatible.

PROOF: All elements of G are v-mappings accepted by acceptors in Q at round k. These v-mappings are guaranteed to be compatible because bA is assumed to be conservative (proof assumption 2).

2.4. $NilExtension(\sqcup G, Proposer)$ is safe at r in bA

PROOF: By the definition of *safe at*, it suffices to

ASSUME: There exist v-mapping w and round $s < r$ such that w is choosable at s in bA

PROVE: $w \sqsubseteq NilExtension(\sqcup G, Proposer)$

2.4.1. CASE: $s < k$

2.4.1.1. Choose $a \in AS$

PROOF: a exists by step 2.2, which states that AS is not empty.

2.4.1.2. $w \sqsubseteq bA_a[k]$

PROOF: By the definition of AS, $bA_a[k] \neq none$. Since bA is safe (proof assumption 1), any v-mapping choosable at a round lower-numbered than k, including w given that step 2.4.1 considers only the case where $s < k$, must be a prefix of $bA_a[k]$.

2.4.1.3. $bA_a[k] \sqsubseteq \sqcup G$

PROOF: By the definition of G and least upper bound, and the fact that $a \in AS$ (step 2.4.1.1).

2.4.1.4. Q.E.D.

PROOF: Steps 2.4.1.2 and 2.4.1.3, and the fact that $\sqsubseteq$ is a partial order relation over v-mappings imply that $w \sqsubseteq \sqcup G$. Moreover, by the definition of *Nil*-extension, $\sqcup G \sqsubseteq NilExtension(\sqcup G, Proposer)$. Since $\sqsubseteq$ is a partial order relation over v-mappings, $w \sqsubseteq NilExtension(\sqcup G, Proposer)$.

2.4.2. CASE: $s \geq k$

LET: P be the set $\{p \in Proposer : pA_p[s] = Nil\}$

2.4.2.1. Choose Q_w such that $w \sqsubseteq NilExtension(bA_a[s], P)$ for every acceptor a in Q_w with $\widehat{bA}_a > s$

PROOF: Q_w exists by the definition of choosable and the assumption of step 2.4.

2.4.2.2. Choose $a \in Q \cap Q_w$

PROOF: a exists by the Quorum Requirement.

2.4.2.3. $bA_a[s] \neq none$

PROOF: Proof assumption 3 states that $\widehat{bA}_a \geq r$ and the assumption of step 2.4 states that $r > s$; therefore $\widehat{bA}_a > s$. Steps 2.4.2.1 and 2.4.2.2 imply that $w \sqsubseteq \mathit{NilExtension}(bA_a[s], P)$, which is not possible if $bA_a[s] = \mathit{none}$.

2.4.2.4. $s = k$

PROOF: If $s > k$, and given that $s < r$ by the assumption of step 2.4, then step 2.4.2.3 above contradicts the definition of k since a belongs to Q, $s > k$ and $bA_a[s] \neq \mathit{none}$.

2.4.2.5. $w \sqsubseteq \mathit{NilExtension}(bA_a[k], P)$

PROOF: Steps 2.4.2.1, 2.4.2.2, and 2.4.2.4.

LET: $P^- \triangleq P \setminus \mathit{Dom}(bA_a[k])$

2.4.2.6. No v-mapping in G maps a proposer in P^- to a value different from *Nil*

PROOF: Assume, for the sake of contradiction, that there is a v-mapping in G that maps an element p of P^- to something different from *Nil*. Since $\langle bA, pA \rangle$ is conservative and $p \notin \mathit{Dom}(bA_a[k])$, $pA_p[k]$ must equal the mapped value, which contradicts the definition of P.

2.4.2.7. $\mathit{NilExtension}(bA_a[k], P) \sqsubseteq \mathit{NilExtension}(\sqcup G, \mathit{Proposer})$

PROOF: Steps 2.4.2.3 and 2.4.2.4 imply that $a \in AS$ and, therefore, $bA_a[k] \in G$. The definition of a *Nil*-extension and step 2.4.2.6 complete the proof.

2.4.2.8. Q.E.D.

PROOF: By steps 2.4.2.5 and 2.4.2.7, and the transitivity of $\sqsubseteq$.

2.4.3. Q.E.D.

2.5. Q.E.D.

PROOF: Directly, since $\mathit{NilExtension}(\sqcup G, \mathit{Proposer})$ is the value returned by $\mathit{ProvedSafe}(Q, r, bA)$ in case KS is not empty.

3. Q.E.D.

PROOF: Since KS is defined to be a set, all cases are being covered.

A.2 Abstract Collision-fast Paxos

Our proof of correctness starts with an abstract algorithm that can be more easily proved correct. It is based upon the following variables:

learned An array of v-mappings, where $learned[l]$ is the v-mapping currently learned by learner l. Initially, $learned[l] = \perp$ for all learners l.

proposed The set of proposed values. It initially equals the empty set.

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$ and $bA_a[r] = none$ for every acceptor a and round r.

pA A proposal array. It represents the proposal history. Initially, $pA_p[r] = none$ for every proposer p and round r.

minTried An array of v-mappings, where $minTried[r]$ is either a v-mapping or equal to *none*, for every round r. Initially, $minTried[0] = \perp$ and $maxTried[r] = none$ for all $r > 0$.

The Abstract Collision-fast Paxos algorithm satisfies the following invariants, which, as we prove next, imply the properties Nontriviality and Consistency of M-Consensus.

minTried **Invariant** For every round r, if $minTried[r] \neq none$, then

1. $minTried[r]$ is proposed.
2. $minTried[r]$ is safe at r in $\langle bA, pA \rangle$.
3. If $minTried[r] \neq \perp$, then $minTried[r]$ is valued and complete.

bA **Invariant** For all acceptors a and rounds r, if $bA_a[r] \neq none$, then

1. $minTried[r] \sqsubseteq bA_a[r]$.
2. $bA_a[r]$ is a valued and proposed v-mapping.
3. If $minTried[r] = \perp$ then $\forall p \in Dom(bA_a[r]) : bA_a[r](p) = pA_p[r]$; otherwise, $bA_a[r] = minTried[r]$.

pA **Invariant** For all proposers p and rounds r, if $pA_p[r] \neq none$, then $pA_p[r]$ is either *Nil* or a proposed value.

learned **Invariant** For every learner l:

1. $learned[l]$ is a nontrivial proposed v-mapping.
2. $learned[l]$ is the lub of a finite set of v-mappings chosen in $\langle bA, pA \rangle$.

Proposition 3 *The* learned *invariant implies the Nontriviality property of M-Consensus.*

PROOF: By part 1 of the *learned* invariant.

Proposition 4 *Invariants* minTried, bA, *and* learned *imply the Consistency property of M-Consensus.*

PROOF: By the definition of Consistency, it suffices to assume that invariants bA and *learned* are true, and prove that, for every pair of learners l_1 and l_2, $learned[l_1]$ and $learned[l_2]$ are compatible. The proof is divided into four steps, presented below:

1. $\langle bA, pA \rangle$ is safe.
   PROOF: This follows from part 1 of the bA invariant, part 2 of the *minTried* invariant, the fact that the extension of a safe v-mapping is also safe by definition, and the definition of a safe ballot array (Definition 12).

LET: $S = \{v : v \text{ is chosen in } \langle bA, pA \rangle\}$

2. S is compatible.
   PROOF: By step 1 and Proposition 1.
3. For every learner l, $learned[l] \sqsubseteq \sqcup S$.
   PROOF: This is true by part 2 of the *learned* invariant and the definition of least upper bound, which implies that if set S is compatible, then the lub of S is equal to or extends the lub of any subset of S.
4. Q.E.D.
   PROOF: By step 3 and the definition of compatible c-structs.

Abstract Multicoordinated Paxos has six atomic actions, described below. A complete specification of the algorithm in TLA$^+$ is given in Section D.

Propose(V) for any value V. It is enabled iff $V \notin proposed$ and sets *proposed* to $proposed \cup \{V\}$.

JoinRound(a, r) for any acceptor a and round r. It is enabled iff $\widehat{bA}_a < r$ and sets $\widehat{bA}_a$ to r.

StartRound(r, Q) for any round r and quorum Q of acceptors. It is enabled iff

- $minTried[r] = none$ and
- $\forall a \in Q : r \leq \widehat{bA}_a$.

It sets $minTried[r]$ to $ProvedSafe(Q, r, bA)$.

Suggest(p, r, V) for proposer p, round r, and (possibly *Nil*) value V, where p is a collision-fast proposer of r. It is enabled iff

- $pA_p[r] = none$ and
- either (i)$minTried[r] \notin \{\bot, none\}$ and $V = minTried[r](p)$, (ii) V is a proposed value, or (iii) $V = Nil$ and there is a collision-fast proposer q of r such that $pA_q[r] \notin \{Nil, none\}$.

It sets $pA_p[r]$ to V.

ClassicVote(a, r, v) for acceptor a, round r, and v-mapping v. Let P^- be the subset of collision-fast proposers of r such that $p \in P \iff pA_p[r] = none$, and let $MaxT$ equal the v-mapping that maps each proposer q in $Proposer \setminus P^-$ to $pA_q[r]$ if q is collision-fast for r or to Nil otherwise. This action is enabled iff

- $\widehat{bA}_a \leq r$,
- v is a valued v-mapping,
- $minTried[r] \neq none$,
- $v \sqsubseteq MaxT$, if $minTried[r] = \bot$, or $v = minTried[r]$, othewise, and
- either $bA_a[r] = none$ or $bA_a[r] \sqsubset v$.

It sets $\widehat{bA}_a$ to r and $bA_a[r]$ to v.

AbstractLearn(l, v) for any learner l and v-mapping v. It is elabled iff v is chosen in $\langle bA, pA\rangle$ and sets $learned[l]$ to $learned[l] \sqcup v$.

The following proposition proves that the algorithm also satisfies the Stability property of Generalized Consensus.

Proposition 5 *Abstract Multicoordinated Paxos satisfies the Stability property of Generalized Consensus.*

PROOF: For any learner l, the only action that changes the value of $learned[l]$ is *AbstractLearn*(l, v). Since, by the definition of lub, this action can only extend the value of $learned[l]$, Stability is ensured.

It is easy to verify that the algorithm's actions keep the type invariant of the variables it uses. The most complicated case concerns the ballot array bA, updated by actions *JoinRound*(a, r) and *ClassicVote*(a, r, v). However, action *JoinRound*(a, r) only increases the value of $\widehat{bA}_a$ and action *ClassicVote*(a, r, v) sets $bA_a[r]$ to v, where r always equals $\widehat{bA}_a$ after the action is executed. These changes to bA keep it a ballot array according to the definition.

It remains to prove that the abstract algorithm satisfies the invariants *minTried*, *bA*, *pA*, and *learned*. For the sake of simplicity, however, we use some extra notation in the proof. When analyzing the execution of an action, we use ordinary expressions such as *exp* to represent the value of that expression before the action is executed, and we let *exp′* be the value of that expression after the action execution.

Proposition 6 *Abstract Multicoordinated Paxos satisfies the invariants* minTried*,* bA*,* pA*, and* learned*.*

PROOF: The invariants are trivially satisfied in the initial state. Therefore, it suffices to assume that the invariants are true and prove that, for every action α, they remain true if α is executed. We do that in the following, analyzing case by case.

1. CASE: Action *Propose*(V) is executed, where V is a non-*Nil* value.
   PROOF SKETCH: Action *Propose*(V) only changes variable *proposed*, which is the set of proposed values, and does that by adding a new element to it. Invariant conditions that do not refer to this set are obviously preserved. The others are kept true since the set *proposed* only increases and v-mappings composed of proposed values remain composed of proposed values.

2. CASE: Action *JoinRound*(a, r) is executed, where a is an acceptor and r is a round number.
   PROOF SKETCH: Action *JoinRound*(a, r) only changes $\widehat{bA}_a$, setting it to r, which is bigger than $\widehat{bA}_a$. Invariant conditions that do not refer to $\widehat{bA}_a$ are obviously preserved. It remains to check that safe or chosen v-mappings in $\langle bA, pA\rangle$ are kept safe or chosen in $\langle bA', pA'\rangle$. The definition of *chosen* does not involve $\widehat{bA}_e$ for any acceptor e. The definition of *safe* is based upon the definition of *choosable at*, which does refer to $\widehat{bA}_e$, but implies that a v-mapping w that is choosable at round k in $\langle bA', pA'\rangle$ is also choosable at k in $\langle bA, pA\rangle$. By the definition of *safe*, this implies that a value x that is safe at round s in $\langle bA, pA\rangle$ is also safe at s in $\langle bA', pA'\rangle$.

3. CASE: Action *StartRound*(r, Q) is executed, where r is a round and Q is a quorum of acceptors.
   PROOF SKETCH: Action *StartRound*(r, Q) changes *minTried*$[r]$ from *none* to *ProvedSafe*(Q, r, bA). The action does not change the other variables.The bA invariant implies that the pair $\langle bA, pA\rangle$ is conservative,

which ensures that $ProvedSafe(Q, r, bA)$ is safe at r in $\langle bA, pA\rangle$. Moreover, the bA invariant states that all accepted v-mappings are proposed and valued, which guarantees that $ProvedSafe(Q, r, bA)$ is either $\bot$ or a proposed, valued, and complete v-mapping by the definition of $ProvedSafe(Q, r, bA)$. All these things imply that the action preserves the *minTried* invariant. It preserves the bA invariant because it does not change bA and $bA_e[r]$ is ensured to equal *none*, for any acceptor e, by the bA invariant itself. The other invariants are obviously preserved because the variables they refer to do not change.

4. CASE: Action $Suggest(p, r, V)$ is executed, where p is a collision-fast proposer for round r and V is either a proposed value or *Nil*.
   PROOF SKETCH: This action only changes $pA_p[r]$ from *none* to V and clearly keeps all invariants.

5. CASE: Action $ClassicVote(a, r, v)$ is executed, where a is an acceptor, r is a round number, and v is a v-mapping.
   PROOF SKETCH: The definition of *choosable at* implies that if a value is choosable at round s in $\langle bA', pA'\rangle$, then it is chossable at s in $\langle bA, pA\rangle$. This fact, by the definition of *safe at*, implies that a value that is safe at round s in $\langle bA, pA\rangle$ is necessarily safe at s in $\langle bA', pA'\rangle$. This, together with the fact that no variable but bA is updated by this action, implies that the action preserves invariants *minTried*, pA, and *learned*. As for the bA invariant, there are two cases to consider. If $minTried[r] \neq \bot$, then this action sets $bA_a[r]$ to $minTried[r]$, which is ensured to be valued, safe, and complete. In this case, the bA invariant is clearly preserved. Now, let us assume $minTried[r] = \bot$ and check if the action preserves the bA invariant with respect to $bA_a[r]$, which is the only entry of bA changed in this action. Condition 1 of the bA invariant is trivially true because $minTried[r] = \bot$. Condition 2 is true because $MaxT$ is proposed by the pA invariant and v is ensured to be valued by the action's pre-condition. Condition 3 is ensured by the definition of $MaxT$.

6. CASE: Action $AbstractLearn(l, v)$ is executed, where l is a learner and v is a v-mapping.
   PROOF SKETCH: Action $AbstractLearn(l, v)$ only changes variable *learned*, which is the array of learned v-mappings, and does that by extending one entry to the lub of it with a chosen v-mapping. Invariants *maxTried* and bA are obviously preserved. The first part of the *learned* invariant is pre-

served because of the pA invariant and the fact that v-mappings accepted by acceptors are both proposed and valued. The second part is obviously preserved.

A.3 Distributed Abstract Collision-fast Paxos

As an intermediate step in our proof, we introduce a distributed version of the abstract algorithm in the previous section. This algorithm is based on the same variables as the previous algorithm plus a variable *msgs*, used to simulate a message passing system by holding the messages sent between agents. Variable initialization is done as before for the common variables, and *msgs* is set to $\{\langle \text{“}\texttt{2S}\text{”}, 0, \bot\rangle\}$ initially, which implies that a $2S$ message for round 0 is implicitly sent when the algorithm starts. Message duplication is implemented by never taking messages out of set *msgs*, which would permanently enable actions that depend on an existing message. Since we are proving only safety, we do not have to implement the loss of messages because a message loss would only imply that some actions would not be executed.

The distributed abstract algorithm is described in terms of the following actions. Its formal specification in TLA$^+$ is given in the appendix section D.

Propose(V) for any value V. It is enabled iff $V \notin$ *proposed*. It sets *proposed* to *proposed* $\cup \{V\}$ and adds message $\langle \text{“}\texttt{propose}\text{”}, V\rangle$ to *msgs*.

Phase1a(c, r) executed by coordinator c, for round r. The action is enabled iff $minTried[r] = none$. It sends the message $\langle \text{“}\texttt{1a}\text{”}, r\rangle$ to acceptors (adds it to *msgs*).

Phase1b(a, r) executed by acceptor a, for round r. The action is enabled iff

- $\widehat{bA}_a < r$ and
- $\langle \text{“}\texttt{1a}\text{”}, r\rangle \in msgs$

It sets $\widehat{bA}_a$ to r and adds message $\langle \text{“}\texttt{1b}\text{”}, r, a, bA_a\rangle$ to *msgs*.

Phase2Start(r) for round r. The action is enabled iff:

- $minTried[r] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle \text{“}\texttt{1b}\text{”}, r, a, \rho\rangle$ in *msgs*, for some ρ.

Let $v = \mathit{ProvedSafe}(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle \text{“}\texttt{1b}\text{”}, r, a, \rho\rangle$ in msgs with $\rho = \beta_a$. This action sets $\mathit{minTried}[r]$ to v and adds message $\langle \text{“}\texttt{2S}\text{”}, r, v\rangle$ to msgs.

Phase2Prepare(p, r) executed by proposer p, for round r. It is enabled iff:

- $pA_p[r] = \mathit{none}$ and
- There exists message $\langle \text{“}\texttt{2S}\text{”}, r, v\rangle$ in msgs with $v \neq \bot$

It sets $pA_p[r]$ to $v(p)$.

Phase2a(p, r, V) executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- p is a collision-fast proposer of r,
- $pA_p[r] = \mathit{none}$,
- $\langle \text{“}\texttt{2S}\text{”}, r, \bot\rangle \in \mathit{msgs}$, and
- either $\langle \text{“}\texttt{propose}\text{”}, V\rangle \in \mathit{msgs}$ or $V = \mathit{Nil}$ and there exists a message $\langle \text{“}\texttt{2a}\text{”}, r, \langle q, U\rangle\rangle$ in msgs with $U \neq \mathit{Nil}$.

This action sets $pA_p[r]$ to V and adds message $\langle \text{“}\texttt{2a}\text{”}, r, \langle p, V\rangle\rangle$ to msgs.

Phase2b(a, r, v) executed by acceptor a, for round r and v-mapping v. It is enabled iff $\widehat{bA}_a \leq r$ and either one of the following conditions hold:

a) $bA_a[r] = \mathit{none}$ and message $\langle \text{“}\texttt{2S}\text{”}, r, v\rangle$ exists in msgs, where $v \neq \mathit{Nil}$, or

b) message $\langle \text{“}\texttt{2a}\text{”}, r, \langle p, V\rangle\rangle$ exists in msgs, where $V \neq \mathit{Nil}$, and either one of the two following conditions hold:

   b1) $bA_a[r] = \mathit{none}$ and $v = \mathit{NilExtension}(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

   b2) $bA_a[r] \neq \mathit{none}$ and $v = bA_a[r] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$ to r and $bA_a[r]$ to v, and adds message $\langle \text{“}\texttt{2b}\text{”}, r, a, v\rangle$ to msgs.

Learn(l, v) executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a"$, r, \langle p, Nil\rangle\rangle \in msgs$ and
- $\forall a \in Q : \langle$"2b"$, r, a, u\rangle \in msgs$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

The distributed abstract algorithm implements the the non-distributed version in the sense that all behaviors of the former are also behaviors of the latter.

Proposition 7 *Distributed Abstract Collision-fast Paxos implements the Abstract Collision-fast Paxos specification.*

PROOF SKETCH: The initial state of both algorithms with respect to their shared variables is exactly the same. As a result, to prove this proposition we must only show that every action in the distributed algorithm implements an action of the non-distributed algorithm with respect to the variable states before and after the action is taken [1]. In the following we analyze each action of the distributed version.

Propose(V) This action clearly implements the action with the same name in the non-distributed algorithm. The only difference has to do with variable *msgs* which is not present in the non-distributed version.

Phase1a(c, r) This action changes only variable *msgs*, and implements a no-action (stuttering) step in the non-distributed algorithm, since it keeps the rest of the state the same as before.

Phase1b(a, r) This action clearly implements action *JoinRound* of the non-distributed algorithm. It is more restrictive, though, since it requires a "1b" message for r to be present in *msgs*.

Phase2Start(r) This action implements action *StartRound* of the non-distributed algorithm. Let Q be the quorum of action *Phase2Start*; the reception of the "1b" messages for round r coming from acceptors a in Q implies that every acceptor $a \in Q$ has set $\widehat{bA}_a$ to r. Since $\widehat{bA}_a$ is never decreased, we can conclude that $\widehat{bA}_a \geq r$ for every acceptor a in Q, as required by action *StartRound*. By the definition of *ProvedSafe* and the fact that the vectors sent in the "1b" messages are consistent with the current state of bA, one can easily verify that *ProvedSafe*(Q, r, β) returns exactly the same value as *ProvedSafe*(Q, r, bA).

Phase2Prepare(p, r) This action implements *Suggest*(p, r, V) when $minTried[r]$ is different from $\bot$ (identified by the "2S" message) and $V = minTried[r](p)$.

Phase2a(p, r, V) This action implements *Suggest*(p, r, V) when *minTried*$[r]$ equals $\perp$ (identified by the received "`2S`" message). Notice that V is either a proposed value (received in a "`propose`" message) or it equals *Nil* but other collision-fast proposer q for round r has set $pA_q[r]$ to a non-*Nil* value, a situation identified by a "`2a`" message.

Phase2b(a, r, v) This action implements *ClassicVote*(a, r, v). There are three cases to consider:

- $bA_a[r] = none$ and message $\langle$"`2S`"$, r, v\rangle$ ($v \neq Nil$) exists in *msgs*, where $v \neq Nil$.

  In this case, *minTried*$[r] = v$ and the implementation of *ClassicVote*(a, r, v) is easily verified.

- message $\langle$"`2a`"$, r, \langle p, V\rangle\rangle$ ($V \neq Nil$) exists in *msgs*, $bA_a[r] = none$, and $v = NilExtension(\perp \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r.

  In this case, since a "`2a`" message was sent and it is only sent by a collision-fast proposer of r when *minTried*$[r] = \perp$, we can infer that p is a collision-fast proposer of r and *minTried*$[r] = \perp$. By the definition of *MaxT* in *ClassicVote*, it is easy to see that v satisfies the pre-condition of this action. The rest of the action implementation is easily checked.

- message $\langle$"`2a`"$, r, \langle p, V\rangle\rangle$ ($V \neq Nil$) exists in *msgs*, $bA_a[r] \neq none$, and $v = bA_a[r] \bullet \langle p, V\rangle$.

  Once again, the existence of a "`2a`" message for round r implies that p is collision-fast for r and *minTried*$[r] = \perp$. Notice that no value $pA_p[r]$ can change after it is set to something different from *none* and the same happens with *minTried*$[r]$. Since entry $bA_a[r]$ is only changed by a *Phase*$2b(a, r, v)$ action, by the definition of *MaxT* it is easy to see that $bA_a[r] \sqsubseteq MaxT$. Given that $v = bA_a[r] \bullet \langle p, V\rangle$, it also follows from the definition of *MaxT* that $v \sqsubseteq MaxT$.

Learn(l, v) This action implements action *AbstractLearn*(l, v) by the definition of a chosen v-mapping and the fact that "`2a`" and "`2b`" messages reflect stable changes, in the sense that no further changes can happen, to entries in pA and bA respectively.

A.4 Collision-fast Paxos

To prove correctness of algorithm presented in Section 4.1, we first add the following history variables to the algorithm presented in the previous section.

prnd An array of round numbers, where $prnd[p]$ represents the current round of proposer p. Initially 0.

pval An array of v-mappings, where $pval[p]$ represents the v-mapping fast-proposed by proposer p on round $prnd[p]$ or *none*, if p has not fast-proposed in that round. Initially *none*.

crnd An array of round numbers, where $crnd[c]$ represents the current round of coordinator c. Initially 0.

cval An array of v-mappings, where $cval[c]$ represents the latest v-mapping sent by coordinator c in a phase "2S" message for round $crnd[c]$. Initially $\perp$ for the coordinator of round 0 and *none* for all the others.

rnd An array of round numbers, where $rnd[a]$ is the current round of acceptor a, that is, the highest-numbered round a has heard of. Initially 0.

vrnd An array of round numbers, where $vrnd[a]$ is the round at which acceptor a has accepted the latest v-mapping. Initially 0.

vval An array of v-mappings, where $vval[a]$ is the v-mapping acceptor a has accepted at $vrnd[a]$ or *none*. Initially *none*.

msgs2 Counterparts of the messages sent by the original protocol, but built with the values of history variables. Initially $\{\langle \text{“2S”}, 0, \perp \rangle\}$.

Propose(V) for any value V. It is enabled iff $V \notin proposed$. It sets *proposed* to $proposed \cup \{V\}$ and adds message $\langle$“propose”, $V\rangle$ to *msgs* and *msgs2*.

Phase1a(c, r) executed by coordinator c, for round r. The action is enabled iff $minTried[r] = none$. It sets $crnd[c]$ to r and $cval[c]$ to *none*, and adds a message $\langle$“1a”, $c, m\rangle$ to *msgs* and *msgs2*.

Phase1b(a, r) executed by acceptor a, for round r. The action is enabled iff

- $\widehat{bA}_a < r$ and

- $\langle$"1a", $r\rangle \in msgs$

It sets $\widehat{bA}_a$ to r and $rnd[a]$ to r and adds the message $\langle$"1b", r, a, $bA_a\rangle$ to $msgs$ and "1b"r,a,vrnd[a],vval[a] to $msgs2$.

Phase2Start(r) executed by the coordinator c of round r, for round r. The action is enabled iff:

- $minTried[r] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $\rho\rangle$ in $msgs$, for some ρ.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", r, a, $\rho\rangle$ in $msgs$ with $\rho = \beta_a$. This action sets $minTried[r]$ and $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S", r, $v\rangle$ to $msgs$ and $msgs2$.

Phase2Prepare(p, r) executed by proposer p, for round r. It is enabled iff:

- $pA_p[r] = none$ and
- There exists message $\langle$"2S", r, $v\rangle$ in $msgs$

If $v \neq \perp$, it sets $pA_p[r]$ and $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\perp$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

Phase2a(p, r, V) executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- p is a collision-fast proposer of r,
- $pA_p[r] = none$,
- $\langle$"2S", $r, \perp\rangle \in msgs$, and
- either $\langle$"propose", $V\rangle \in msgs$ or $V = Nil$ and there exists a message $\langle$"2a", $r, \langle q, U\rangle\rangle$ in $msgs$ with $U \neq Nil$.

This action sets $pA_p[r]$ and $pval[p]$ to V and adds message $\langle$"2a", $r, \langle p, V\rangle\rangle$ to $msgs$ and $msgs2$.

Phase2b(a, r, v) executed by acceptor a, for round r and v-mapping v. It is enabled iff $\widehat{bA}_a \leq r$ and either one of the following conditions hold:

a) $bA_a[r] = none$ and message $\langle$"2S", $r, v\rangle$ exists in $msgs$, where $v \neq Nil$, or

b) message $\langle$"2a", $r, \langle p, V\rangle\rangle$ exists in *msgs*, where $V \neq Nil$, and either one of the two following conditions hold:

b1) $bA_a[r] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $bA_a[r] \neq none$ and $v = bA_a[r] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to r, $bA_a[r]$ and $vval[a]$ to v, and adds message $\langle$"2b", $r, a, v\rangle$ to *msgs* and *msgs2*.

Learn(l, v) executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", $r, \langle p, Nil\rangle\rangle \in msgs$ and
- $\forall a \in Q : \langle$"2b", $r, a, u\rangle \in msgs$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

Variables *prnd*, *pval*, *crnd*, *cval*, *rnd*, *vrnd*, *vval*, and *msgs2* appear in no pre-condition and, therefore, are clearly history variables satisfying conditions H1-5 of [1]. This implies that the resulting algorithm is equivalent to (i.e., accepts the same behaviors as) the previous one without such variables. The following invariants can be easily proved for this new algorithm:

InvDA1: $crnd[c] = k \Rightarrow \forall j > k : c$ is coordinator of $j : minTried[j] = none$

InvDA2: $minTried[crnd[c]] = cval[c]$

InvDA3: $rnd[a] = \widehat{bA}_a$

InvDA4: $vrnd[a] = k \iff \begin{array}{ll} \wedge & bA_a[k] \neq none \\ \wedge & \forall j > k : bA_a[j] = none \end{array}$

InvDA5: $vval[a] = bA_a[vrnd[a]]$

InvDA6: $prnd[p] = k \Rightarrow \forall j > k : pA_p[j] = none$

InvDA6.5: $pval[p] = pA_p[prnd[p]]$

InvDA7: $\langle$"1a", $m\rangle \in msgs \iff \langle$"1a", $m\rangle \in msgs2$

InvDA8: $\langle$"1b", $m, \rho\rangle \in msgs \iff \langle$"1b", $m, vval, vrnd\rangle \in msgs2$, where $vrnd$ is the highest balnum k such that $\rho[k] \neq none$ and $vval$ equals $\rho[vrnd]$.

InvDA9: $\langle$"2S", $m, v\rangle \in msgs \iff \langle$"2S", $m, v\rangle \in msgs2$

InvDA10: $\langle$"2a", $m, v\rangle \in msgs \iff \langle$"2a", $m, v\rangle \in msgs2$

InvDA11: $\langle$"2b", $m, v\rangle \in msgs \iff \langle$"2b", $m, v\rangle \in msgs2$

We can use these invariants to rewrite the pre-conditions of the previous algorithm's actions in the following way:

Propose(V) for any value V. It is enabled iff $V \notin proposed$. It sets *proposed* to $proposed \cup \{V\}$ and adds message $\langle$"propose", $V\rangle$ to *msgs* and *msgs2*.

The action remains the same.

Phase1a(c, r) executed by coordinator c, for round r. The action is enabled iff

- c is the coordinator of round r and
- $crnd[c] \leq r$.

It sets $crnd[c]$ to r and $cval[c]$ to *none*, and adds a message $\langle$"1a", $c, m\rangle$ to *msgs* and *msgs2*.

By invariant InvDA1.

Phase1b(a, r) executed by acceptor a, for round r. The action is enabled iff

- $rnd[a] < r$ and
- $\langle$"1a", $r\rangle \in msgs2$

It sets $\widehat{bA}_a$ and $rnd[a]$ to r, and adds the message $\langle$"1b", $r, a, bA_a\rangle$ to *msgs* and "1b"r,a,vrnd[a],vval[a] to *msgs2*.

By invariants InvDA3 and InvDA7.

Phase2Start(r) executed by the coordinator c of round r, for round r. The action is enabled iff:

- $crnd[c] = r$
- $cval[c] = none$ and

- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $vval$, $vrnd\rangle$ in $msgs2$.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", r, a, $vrnd$, $vval\rangle$ in $msgs2$ with $\beta_a vrnd = vval$ and $\beta_a or = none$ for any round $or \neq vrnd$. This action sets $minTried[r]$ and $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S", r, $v\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA2 and InvDA8. $ProvedSafe(Q, r, \beta)$ still gives the expected result because it only uses the latest values accepted by each acceptor, the only value in β.

Phase2Prepare(p, r) executed by proposer p, for round r. It is enabled iff:

- $prnd[p] \leq r$ and
- There exists message $\langle$"2S", r, $v\rangle$ in $msgs2$

If $v \neq \bot$, it sets $pA_p[r]$ and $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\bot$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

By invariants InvDA6, Inv6.5, and InvDA9. If $v \neq \bot$, then this action implements its previous version. Otherwise, it implements a stuttering step of its previous specification.

Phase2a(p, r, V) executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- $prnd[p] = r$
- p is a collision-fast proposer of r,
- $pval[r] = none$,
- $\langle$"2S", r, $\bot\rangle \in msgs2$, and
- either $\langle$"propose", $V\rangle \in msgs2$ or $V = Nil$ and there exists a message $\langle$"2a", r, $\langle q, U\rangle\rangle$ in $msgs2$ with $U \neq Nil$.

This action sets $pA_p[r]$ and $pval[p]$ to V and adds message $\langle$"2a", r, $\langle p, V\rangle\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA6 and InvDA6.5. Since it is more restrictive than its previous version (requires the prior action to execute), and the other pre-conditions are equivalent, the action implements its previous version.

Phase2b(a, r, v) executed by acceptor a, for round r and v-mapping v. It is enabled iff $rnda \leq r$ and either one of the following conditions hold:

a) $vrnd[a] < r \vee vval[a] = none$ and message $\langle$"2S", $r, v\rangle$ exists in $msgs2$, where $v \neq Nil$, or

b) message $\langle$"2a", $r, \langle p, V\rangle\rangle$ exists in $msgs2$, where $V \neq Nil$, and either one of the two following conditions hold:

b1) $vrnd[a] < r \vee vval[a] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $vrnd[a] = r \wedge vval[a] \neq none$ and $v = vval[a] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to r, $bA_a[r]$ and $vval[a]$ to v, and adds message $\langle$"2b", $r, a, v\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA4, InvDA5, InvDA9, InvDA10, and because $rnd[a] \geq vrnd[a]$ (By InvDA3, InvDA4 and the definition of ballot array).

Learn(l, v) executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", $r, \langle p, Nil\rangle\rangle \in msgs2$ and
- $\forall a \in Q : \langle$"2b", $r, a, u\rangle \in msgs2$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

By invariant InvDA11.

The resulting algorithm now has variables bA, pA, *minTried* and *msgs* as history variables, since they do not appear on any action's pre-condition and are only updated. This algorithm is, therefore, equivalent to one that does not contain such variables, which we present below.

Propose(V) for any value V. It is enabled iff $V \notin proposed$. It sets *proposed* to $proposed \cup \{V\}$ and adds message $\langle$"propose", $V\rangle$ to *msgs2*.

Phase1a(c, r) executed by coordinator c, for round r. The action is enabled iff

- c is the coordinator of round r and
- $crnd[c] \leq r$.

It sets $crnd[c]$ to r and $cval[c]$ to *none*, and adds a message $\langle$"1a", $c, m\rangle$ to *msgs2*.

Phase1b(a, r) executed by acceptor a, for round r. The action is enabled iff

- $rnd[a] < r$ and
- $\langle$"1a"$, r\rangle \in msgs2$

It sets $rnd[a]$ to r and adds the message "1b"r,a,vrnd[a],vval[a] to *msgs2*.

Phase2Start(r) executed by the coordinator c of round r, for round r. The action is enabled iff:

- $crnd[c] = r$
- $cval[c] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b"$, r, a, vval, vrnd\rangle$ in $msgs2$.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b"$, r, a, vrnd, vval\rangle$ in $msgs2$ with $\beta_a vrnd = vval$ and $\beta_a or = none$ for any round $or \neq vrnd$. This action sets $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S"$, r, v\rangle$ to $msgs2$.

Phase2Prepare(p, r) executed by proposer p, for round r. It is enabled iff:

- $prnd[p] \leq r$ and
- There exists message $\langle$"2S"$, r, v\rangle$ in $msgs2$

If $v \neq \perp$, it sets $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\perp$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

Phase2a(p, r, V) executed by proposer p, for round r and (possibly *Nil*) value V. The action is enabled iff:

- $prnd[p] = r$
- p is a collision-fast proposer of r,
- $pval[r] = none$,
- $\langle$"2S"$, r, \perp\rangle \in msgs2$, and
- either $\langle$"propose"$, V\rangle \in msgs2$ or $V = Nil$ and there exists a message $\langle$"2a"$, r, \langle q, U\rangle\rangle$ in $msgs2$ with $U \neq Nil$.

This action sets $pval[p]$ to V and adds message $\langle \text{“2a”}, r, \langle p, V \rangle \rangle$ to $msgs2$.

Phase2b(a, r, v) executed by acceptor a, for round r and v-mapping v. It is enabled iff $rnda \leq r$ and either one of the following conditions hold:

a) $vrnd[a] < r \vee vval[a] = none$ and message $\langle \text{“2S”}, r, v \rangle$ exists in $msgs2$, where $v \neq Nil$, or

b) message $\langle \text{“2a”}, r, \langle p, V \rangle \rangle$ exists in $msgs2$, where $V \neq Nil$, and either one of the two following conditions hold:

b1) $vrnd[a] < r \vee vval[a] = none$ and $v = NilExtension(\bot \bullet \langle p, V \rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $vrnd[a] = r \wedge vval[a] \neq none$ and $v = vval[a] \bullet \langle p, V \rangle$.

The action sets $rnd[a]$ and $vrnd[a]$ to r, $vval[a]$ to v, and adds message $\langle \text{“2b”}, r, a, v \rangle$ to $msgs2$.

Learn(l, v) executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle \text{“2a”}, r, \langle p, Nil \rangle \rangle \in msgs2$ and
- $\forall a \in Q : \langle \text{“2b”}, r, a, u \rangle \in msgs2$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

The algorithm presented in Section 4.1 is a stricter implementation of the algorithm above, which can be easily verified by simply comparing their actions. This concludes the proof that Collision-Fast Paxos satisfies the safety requirements of M-Consensus. □

B The Liveness of Collision-fast Paxos

We now prove that the extended Collision-fast Paxos algorithm presented in Section 4.2 satisfies the Liveness property of M-Consensus, given that its liveness condition is eventually satisfied.

Proposition 8 *If there exist proposer p, learner l, coordinator c, and quorum Q, such that $LA(p, l, c, Q)$ holds from some time t_0 on, then eventually $learned[l]$ is complete.*

PROOF: The proof is divided into following steps:

1. No coordinator other than c executes any action after t_0
   PROOF SKETCH: The extended algorithm states that coordinators only execute actions if they believe to be the leader and the definition of $LA(p, l, c, Q)$ states that only c believes to be the leader after t_0.

2. There is a time $t_1 \geq t_0$ after which $crnd[c]$ does not change
   PROOF SKETCH: $crnd[c]$ can only be changed by action $Phase1a$. In the extended algorithm, though, this can only happen if c receives a special message informing about a higher-numbered round already started or if not all collision-fast proposers for $crnd[c]$ are in $activep[c]$. As for the first condition, step 1 implies there is only a finite number of (possibly higher-numbered) rounds started before t_0. As for the second one, the definition of LA states that $activep[c]$ contains only nonfaulty processes after t_0 and no element is taken out of the set. In the extended algorithm, if c starts a new round r after t_0, it is guaranteed that its collision-fast proposers are in $activep[c]$, which makes sure the second condition will not trigger the execution of action $Phase1a$ more than once after t_0.

3. There is a time $t_2 \geq t_1$ after which action $Phase2Start(c, crnd[c])$ will have been executed
   PROOF SKETCH: Let us assume, for the sake of contradiction, that c never executes action $Phase2Start(c, crnd[c])$. By step 2 and the extended algorithm's specification, coordinator c keeps re-sending the "`1a`" message for round $crnd[c]$ to all acceptors. Acceptors in Q do not crash after t_0 by the definition of LA and must receive such messages. If they all execute action $Phase1b$ for round $crnd[c]$, then they will keep re-sending their $1b$ messages and c will eventually execute $Phase2Start$, contradicting our assumption. Therefore, there must be an acceptor $a \in Q$ such that $rnd[a] > crnd[c]$, which prevents the execution of action $Phase1b(a, crnd[c])$. However, in the extended algorithm a would send an infinite number of special messages to c, indicating that a round higher than $crnd[c]$ has been started and this would eventually lead c to execute action $Phase1a$ for a higher-numbered round, contradicting step 2.

4. From t_2 on, $cval[c]$ does not change
   PROOF SKETCH: By steps 2 and 3 and the algorithm's specification.

5. Eventually l learns a complete v-mapping
   By step 4, there are two cases to consider after t_2:

5.1. CASE: $cval[c] = \perp$

5.1.1. From t_2 on, $prnd[q] \leq crnd[c]$ for any proposer q such that q does not crash after t_2
PROOF SKETCH: Assume $prnd[q] > crnd[c]$ after t_2, for some proposer q that does not crash after t_2. In the extended algorithm, by steps 2 and 3, coordinator c keeps sending the "`2S`" message for round $crnd[c]$ to the set of proposers. As a result, q will keep replying to c with special messages indicating that a round higher-numbered than $crnd[c]$ has been started and this will force c to start an even higher-numbered round. This contradicts step 2.

5.1.2. There is a time $t_3 \geq t_2$ after which all proposers q that do not crash after t_3 will have executed action $Phase2Prepare(q, crnd[c])$ and set $prnd[q]$ to $crnd[c]$ and $pval[q]$ to *none*
PROOF SKETCH: By steps 2 and 3, coordinator c keeps sending the "`2S`" message for round $crnd[c]$ and, by step 5.1.1 and the definition of action $Phase2Prepare$, every nonfaulty proposer q must eventually execute action $Phase2Prepare(q, crnd[c])$ based on this "`2S`" message. By the action's definition, $prnd[q]$ is set to $crnd[c]$ and, since the message carries v-mapping $\perp$ (as the value for $cval[c]$), $pval[q]$ is set to *none*.

5.1.3. There is a time t_4 after which some collision-fast proposer q for round $crnd[c]$ will have executed action $Phase2a(q, crnd[c], V)$, where V is a proposed value
PROOF SKETCH: By steps 5.1.1 and 5.1.2 and the definition of LA, each collision-fast proposer of $crnd[c]$ is eventually prepared to execute action $Phase2a$ for round $crnd[c]$ triggered by a "`propose`" or "`2a`" message. Since a "`2a`" message is only sent by action $Phase2a$, some "`propose`" message must trigger the first execution of action $Phase2a$ for round $crnd[c]$. The existence of such a "`propose`" message is guaranteed by steps 5.1.1 and 5.1.2 since they ensure that proposer p (from the definition of LA) will eventually keep sending its "`propose`" message to some collision-fast proposer of round $crnd[c]$. Because the first $Phase2a$ action executed for round $crnd[c]$ is necessarily triggered by a "`propose`" message, its parameter V is a proposed value by the action's definition.

5.1.4. From t_4 on, $rnd[a] \leq crnd[c]$ for any acceptor a such that a does not crash after t_2
PROOF SKETCH: Assume $rnd[a] > crnd[c]$ after t_4, for some acceptor a that does not crash after t_4. By steps 5.1.1 and 5.1.3 and the definition of LA, at least one nonfaulty collision-fast proposer of $crnd[c]$ will eventually keep sending a "`2a`" message for round $crnd[c]$ to a. As a result, a will keep sending notification messages to c indicating that a round higher-numbered than $crnd[c]$ has been started. This would force c to start a new higher-numbered round, contradicting step 2.

5.1.5. There is a time $t_5 \geq t_4$ after which all collision-fast proposers of round $crnd[c]$ will have executed action $Phase2a$ for round $crnd[c]$
PROOF SKETCH: Assume there is a collision-fast proposer q for round $crnd[c]$ such that q never executes action $Phase2a$ for round $crnd[c]$. By steps 5.1.1 and 5.1.2 and the definition of LA, q is eventually prepared to execute action $Phase2a$ for round $crnd[c]$ triggered by a "`propose`" or "`2a`" message. By steps 5.1.1 and 5.1.3, at least one nonfaulty collision-fast proposer of $crnd[c]$ will eventually keep sending a "`2a`" message for $crnd[c]$ to q, which must eventually trigger its execution of action $Phase2a$ for round $crnd[c]$. This contradicts our initial assumption that q does not execute action $Phase2a$ for $crnd[c]$.

5.1.6. There is a time $t_6 \geq t_5$ after which, if action $Phase2a(q, crnd[c], V)$ has been executed for any proposer q and non-Nil value V, then all acceptors in Q will have executed action $Phase2b$ for round $crnd[c]$ triggered by the "`2a`" message sent by q
PROOF SKETCH: By step 5.1.1, if action $Phase2a(q, crnd[c], V)$ is executed, q will keep sending "`2a`" messages to the acceptors. By step 5.1.4, acceptors will be able to execute action $Phase2a$ for any "`2a`" message for round $crnd[c]$ with non-Nil values. Since LA ensures that all collision-fast proposers of $crnd[c]$ are nonfaulty, the "`2a`" message from q will be eventually received by the acceptors in Q (also nonfaulty by LA) and trigger the execution of a $Phase2a$ action for round $crnd[c]$.

5.1.7. Q.E.D.

PROOF SKETCH: After t_6, by steps 5.1.6 and 5.1.4, acceptors in Q keep sending "2b" messages to the learners with the same v-mapping v. By the definition of action $Phase2b$, v maps each proposer that is not collision-fast for $crnd[c]$ to Nil. Moreover, according to step 5.1.6 and the definition of action $Phase2b$, v maps each proposer q that is collision-fast for r and has executed action $Phase2a(q, crnd[c], V)$, where $V \neq Nil$, to V. All other proposers are not mapped by v. According to steps 5.1.1 and 5.1.5, proposers that have executed action $Phase2a$ for round $crnd[c]$ and value Nil keep sending their "2a" messages to the learners. The "2b" messages from the acceptors and the "2a" messages from the collision-fast proposers allow l to eventually learn a complete v-mapping.

5.2. CASE: $cval[c] \neq \bot$

5.2.1. $cval[c]$ is complete
By the definition of action $Phase2Start$ and the fact that $cval[c] \neq \bot$.

5.2.2. From t_2 on, $rnd[a] \leq crnd[c]$ for any acceptor a such that a does not crash after t_2
PROOF SKETCH: Assume $rnd[a] > crnd[c]$ after t_2, for some acceptor a that does not crash after t_2. By steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ to the set of acceptors. In the extended algorithm, though, a will keep replying to c with special messages indicating that a round higher-numbered than $crnd[c]$ has been started and this would force c to start an even higher-numbered round, which contradicts step 2.

5.2.3. Eventually all acceptors a in Q execute action $Phase2b(a, r)$ and set $vrnd[a]$ to $crnd[c]$ and $vval[a]$ to $cval[c]$
PROOF SKETCH: By steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ and, by step 5.2.2, all acceptors in Q must eventually execute action $Phase2b(a, crnd[c])$ based on this "2S" message.

5.2.4. Q.E.D.
PROOF SKETCH: By steps 5.2.2 and 5.2.3, all acceptors in Q will eventually keep sending "2b" messages for round $crnd[c]$

with a complete v-mapping, that is, $cval[c]$. Learner l will eventually receive such messages and learn this complete v-mapping.

6. Q.E.D.

C Collision-fast Atomic Broadcast

In this section we prove that our atomic broadcast protocol indeed satisfies the safety and liveness properties stated in Section 2. We start by presenting the complete specification of the protocol and then proceed with the proofs.

C.1 Collision-Fast Atomic Broadcast

In Section 5.2, we have presented our Collision-Fast Atomic Broadcast algorithm. This protocol uses infinitely many Collision-Fast Paxos instances (Section 4.2), each of them identified by a natural number i and referred as $CFP(i)$ in the protocol. Actions and variables specific of an instance i are identified by the prefix $CFP(i)!$ (instead of the superscript of Section 5.2). The protocol forces the Collision-Fast Paxos instances to execute the same rounds at the same time. As a consequence, some of their variables, namely, *proposed*, *rnd*, *prnd*, *crnd*, and *activep*, are always equal. Intead of keeping multiple copies of these variables, we let all instances share the same copy (and drop the prefix $CFP(i)!$ to simplify the notation). The protocol introduces no other variable.

All actions of the algorithm execute the homonymous action either in one of the Collision-Fast Paxos instances, or in all of them at once in a composed manner. All composed actions pre-conditions are defined only over shared variables and, therefore, either all actions in the composition are enabled or all are disabled. The only exception is action $Phase1a$.

Action $CFP(i)!Phase1a$, for some Collision-Fast Paxos instance i, has one pre-condition over $CFP(i)!msgs$. We define $NewPhase1a$ as a replacement to $CFP(i)!Phase1a$ that changes the said pre-condition to be satisfied if true for $CFP(j)!msgs$, for any instance j. $NewPhase1a$ and the other actions of the algorithm are defined bellow.

Propose(V) Executed to propose a message V. It is the composition of action $CFP(i)!Propose(V)$ for all Collision-Fast Paxos instances i. Logically, each instance sends the message $\langle$"`propose`", $V\rangle$. Since they are all the same, they can be replaced by a single message valid for all Collision-Fast Paxos instances.

*NewPhase*1*a*(i, c, r) Executed by coordinator c to start round r in instances i. The action executes iff:

- c believes itself to be the leader,
- c is the coordinator of round r,
- $crnd[c] \prec r$,
- either c received some special message informing of a round j ($r > j > crnd[c]$) was initiated in any M-Consensus instance, *or* the set of collision-fast proposers of round $crnd[c]$ is not a subset of $activep[c]$.

The action sets $crnd[c]$ to r and $CFP(i)!cval[i]$ to *none*, and sends a message $\langle$"`1a`"$, r\rangle$ in this instance.

*Phase*1*a*(c, r) Executed by coordinator c to start round r. It is the composition action *NewPhase*1*a*(i, c, r) for all Collision-Fast Paxos instances i, where action *NewPhase*1*a*(i, c, r) is defined previously. Logically, each instance sends its own "`1a`" message but, since they are all equal, a single message is sent instead.

*Phase*1*b*(a, r) Executed by acceptor a on round r when it receives the message $\langle$"`1a`"$, r\rangle$. It is the composition of action $CFP(i)!$*Phase*1*b*(a, r) for all Collision-Fast Paxos instances i. Each instance sends its own "`1b`" message but they are all bundled together in a single composite message.

*Phase*2*Start*(c, r) Executed by the coordinator c of round r when it receives a composite "`1b`" message. It is the composition of action $CFP(i)!$*Phase*2*Start*(c, r) for all Collision-Fast Paxos instances i. c sends a "`2S`" message in each instance, but these messages are bundled together in a single physical message. Since only a finite number of instances send "`2S`" messages different from $\langle$"`2S`"$, r, \perp\rangle$, the composite message has a finite size.

*Phase*2*Prepare*(p, r) Executed by proposer p when it receives a composite "`2S`" message for round r. It is the composition of the actions $CFP(i)!$*Phase*2*Prepare*(p, r) of all Collision-Fast Paxos instances i.

*Phase*2*a*(p, r, V) Executed by proposer p to fast-propose message V on round r. It is executed iff p has not fast-proposed V on any Collision-Fast Paxos instance before. The action proposes V on the smaller

Collision-Fast Paxos instance i it is allowed to propose (those instances j such that $CFP(j)!pval[p] = none$); it does so by executing action $CFP(i)!Phase2a(p, r, V)$.

$Phase2b(a, r)$ Executed by acceptor a to accept some v-mapping on some instance i. It does so by executing action $CFP(i)!Phase2b(a, r)$.

$Learn(l, v)$ Executed by learner l to learn the v-mapping v in some Collision-Fast Paxos instance i. The action executes $CFP(i)!Learn(l, v)$.

The sequence of messages delivered by a learner l, $delivered[l]$, is a function of the v-mappings learned by l. To define $delivered[l]$ properly, we assume a total order $<_P$, on the set of proposer agents, and a function $R_P(e, S)$, that gives the rank of an element e of a set S with respect to the other elements in S, according to the order $<_P$ (e.g., $R_P(3, \{3, 1, 5\}) = 2$). The recursive definition of *delivered* is as follows.

$$
\begin{array}{l}
delivered[l] \triangleq \\
\quad \text{LET } defSet(m, s) \triangleq \\
\qquad \{p \in Dom(m) : \\
\qquad\qquad \land\ m[p] \neq Nil \\
\qquad\qquad \land\ \neg\exists i \in 1..Len(s) : s[i] = m[p] \\
\qquad\qquad \land\ \forall q \in Dom(m) : q <_P p \Rightarrow m[q] \neq m[p] \\
\qquad\qquad \land\ \forall q \in Proposer : q <_P p \Rightarrow q \in Dom(m)\} \\
\\
\qquad defSeq(m, s) \triangleq \langle e_1, e_2, \ldots, e_n \rangle : \\
\qquad\quad \land\ n = |defSet(m, s)| \\
\qquad\quad \land\ \forall p \in defSeq(m, s) : R_P(p, defSet(m, s)) = i \Leftrightarrow e_i = m[p] \\
\\
\qquad deliver(l, i, s) \triangleq \\
\qquad\quad \text{IF } Domain = Dom(xlearned[i][l]) \\
\qquad\quad \text{THEN } deliver(l, i + 1, s \circ defSeq(xlearned[i][l], s)) \\
\qquad\quad \text{ELSE } s \circ defSeq(xlearned[i][l], s) \\
\\
\quad \text{IN } deliver(l, 0, \langle\rangle)
\end{array}
$$

Informally, for each learner l an iteration over the M-Consensus instances, from 0 to the smaller one l has not learned a complete mapping yet, builds the sequence $delivered[l]$. In each instance, the iteration proceeds over proposers, ascendingly in the total order $<_P$, adding the mapped value of each proposer to the sequence if it has not been added before. For the instance

with incomplete v-mapping, the iteration proceeds only until a non-mapped proposer is found.

C.2 Safety

In the previous section we described the all the actions of Collision-Fast Atomic Broadcast. As aforementioned, except for one action, all the others simply execute the actions of its Collision-Fast Paxos instances. The exception is action $Phase1a$, that executes $NewPhase1a$ instead of the $Phase1a$ of Collision-Fast Paxos, defined in Section 4. However, comparing the two definitions, it is easy to see that the first is in fact a more restricted version of the latter, and therefore implements it. The proposition below formally states this property.

Proposition 9 *For any Collision-Fast Paxos instance i, coordinator c, and round r, $NewPhase1a(i, c, r)$ implements $CFP(i)!Phase1a(c, r)$.*

ASSUME: There exist a natural number i, a coordinator c and a round number r such that $NewPhase1a(i, c, r)$ is enabled.
PROVE: $CFP(i)!Phase1a(c, r)$ is enabled.

PROOF: Since all pre-conditions of $CFP(i)!Phase1a(c, r)$ are also present in $NewPhase1a(i, c, r)$ and they are all satisfied by assumption, the action $CFP(i)!Phase1a(c, r)$ must also be enabled. □

Proposition 9 implies that all Collision-Fast Paxos instances used in Collsion-Fast Atomic Broadcast satisfy their safety properties. Before showing that Collision-Fast Atomic Broadcast also satisfies its safety properties, we prove some properties of the *delivered* variable.

Proposition 10 *At the initial state, $delivered[l] = \langle\rangle$, for any learner l.*

PROOF: Since *Init* clearly implies the initial state of the M-Consensus instances, for any learner l and instance i, $CFP(i)!learned[l] = \bot$; by the definition of *delivered*, $delivered[l] = \langle\rangle$. □

Proposition 11 *For any learner l, $delivered[l]$ never contains any duplicate.*

PROOF: The proposition is a direct consequence of the definition of *defSet* in *delivered*, that excludes proposers mapped to repeated values. Since only the values mapped by proposers in *defSet* are added to $delivered[l]$, for any learner l, $delivered[l]$ has no duplicates. □

Proposition 12 *For any learner l, $delivered[l]$ contains only broadcast messages.*

PROOF: The initial state of Collision-Fast Atomic Broadcast clearly implies the initial state of all of its M-Consensus instances. These instances therefore satisfy the Nontriviality property of M-Consensus.
From the Nontriviality property of M-Consensus, any value $V \neq Nil$ in the decided mapping of an instance must have been proposed in that instance. By the specification of Collision-Fast Atomic Broadcast, a message is broadcast in Collision-Fast Atomic Broadcast by proposing it in all of its M-Consensus instances in action *Propose*; this is the only action that proposes something. Therefore, any value $V \neq Nil$ decided in one of these instances must be a broadcast message.
By the definition of *delivered*, for any learner l, $delivered[l]$ is formed by non-Nil values decided in some M-Consensus instance. Hence, it is formed by proposed values/broadcast messages. □

Proposition 13 *For any learner l, $delivered[l]$ contains only broadcast messages and no duplicates.*

PROOF: The proposition is true in the initial state, as trivially implied by proposition 10. Properties 11 and 12 imply the proposition in the subsequent states. □

Proposition 14 *For any learner l, if $delivered[l] = s$ at some time, then $s \sqsubseteq delivered[l]$ at all later times.*

PROOF: For some learner l, let k be the smallest instance for which l learned an incomplete v-mapping at some point in time t. For all instances $i < k$, l has learned a complete v-mapping for i (i.e., $CFP(i)!learned[l]$ is complete). Because of the Stability property of M-Consensus, for any instances $i < k$ at any instant $t' > t$, $CFP(i)!learned[l]$ will equal its value at instant t. As for instance k, $CFP(k)!learned[l]$ at time t will be a prefix of its value at time t'.
By its definition, $delivered[l]$ is built by an ascending iteration over the instances i, $0 \leq i \leq k$. For each instance i, the procedure iterates over the proposers p that have been mapped to some value in $CFP(i)!learned[l]$, in the ascending order defined by $<_P$, and appends $V = CFP(i)!learned[l][p]$ to the sequence being created if V is not in the sequence yet. On instance k, the iteration proceeds until the bigger proposer q for which all the smaller proposers have been mapped.

Given the determinism in the iteration and the observations in the first paragraph, the construction of $delivered[l]$ at any instant $t' > t$ will proceed over instances 0 to $k-1$ and then in instance k exactly as at the instant t, building the same sequence, and then possibly extend it with other values mapped in k and bigger instances. Hence, $delivered[l]$ at instant t is a prefix of $delivered[l]$ at any instant $t' > t$. □

Proposition 15 *For any pair of learners $l1$ and $l2$, either $delivered[l1] \sqsubseteq delivered[l2]$ or $delivered[l2] \sqsubseteq delivered[l1]$.*

PROOF: For any two learners l_1 and l_2 let k_i, $i \in \{1, 2\}$, be the smallest M-Consensus instance for which l_i has not learnt a complete v-mapping. Without loss of generality, let l_1 and l_2 be such that length of $delivered[l_1]$ is smaller or equal to the length of $delivered[l_1]$. Clearly, $k_1 \leq k_2$.
By the Consistency property of M-Consensus, for all instances $k \leq k_1$, $CFP(k)!learned[l_1] = CFP(k)!learned[l_2]$. Therefore, by the definition of *delivered*, they induce the same prefix pre-k in $delivered[l_1]$ and $delivered[l_2]$. So it is now enough to show that the values in $CFP(k_1)!learned[l_1]$, that complement pre-k to $delivered[l_1]$, are also a prefix of the complement of pre-k to $delivered[l_2]$.
Let $defSet_i$ equal *defSet*, as in the definition of *delivered*, when evaluating $defined(CFP(k_i)!learned[l_i])$. Because the length of $delivered[l_1]$ is smaller or equal to the length of $delivered[l_2]$, $|defSet_1| \leq |defSet_2|$ and, by implication, $defSet_1 \subseteq defSet_2$. Hence, all values mapped in $CFP(k_1)!learned[l_1]$ that are in $delivered[l_1]$ are also in $delivered[l_2]$, and the first is a sub-sequence of the latter. □

C.3 Liveness

In this section we prove that messages broadcast using the Collision-Fast Atomic Broadcast protocol are eventually delivered. In other words, we want to prove the following proposition.

Proposition 16 *For any proposer p and learner l, if p, l and a quorum of acceptors are nonfaulty and p broadcasts a message m, then eventually $delivered[l]$ contains m.*

ASSUME: There is a proposer p, a learner l, a coordinator c, a quorum Q, and a time t_0 after which $LA(p, l, c, Q)$ always holds.
PROVE: Every message broadcast by p is eventually delivered by l

PROOF: Let c_0 be the coordinator such that, at time t_0, $crnd[c_0]$ is bigger than $crnd[d]$ for any other coordinator d. No message with a round number r bigger than $crnd[c_0]$ could have been sent at instant t_0 (otherwise, $crnd[d] > crnd[c_0]$, for the coordinator d of round r at instant t_0).

1. PROVE: No coordinator other than c executes any action after t_0.
   By the definition of $LA(p, l, c, Q)$, after t_0, no coordinator except for c will ever believe itself to be to leader again. Since this is a pre-condition for all coordinator actions, none will be executed after t_0.
2. PROVE: there is a time $t_1 > t_0$ after which $crnd[c]$ does not change.
   The value of $crnd[c]$ is only changed by executing action $NewPhase1a$ for c. By its defintion, this action can be executed only when c receives a special message informing about a round numbered higher than $crnd[c]$ already started, or if the set of collision-fast proposers for round $crnd[c]$ is not a subset of $activep[c]$.
   Step 1 implies there is just a finite number of (possibly higher-numbered) rounds started before t_0 and that could satisfy the first condition. As for the second condition, the definition of LA states that $activep[c]$ contains only correct processes after t_0 and no element is taken out of the set. Therefore, if c starts a new round r after t_0, it is guaranteed that its collision-fast proposers are in $activep[c]$, which makes sure the second condition will not trigger the execution of action $NewPhase1a$ and, consequently, $Phase1a$ for a higher-numbered round.
3. Q.E.D.
   PROOF: By the steps 1 and 2, by the time t_1 c will have started a round r bigger than any other started and will not start any bigger one. Moreover, all collision-fast proposers of r are correct and accessible. For each Collision-Fast Paxos instance, this situation is equivalent to the one in which its $Phase1a$ has been successfully executed for r and $LA(p, l, c, Q)$ holds. As we have shown in Section B, Collision-Fast Paxos satisfy the liveness property of M-Consensus under these conditions.
   Hence, all Collision-Fast Paxos instances in the atomic broadcast protocol satisfy such property. Therefore, if p proposes a message V, it will eventually be seen by some collision-fast proposer q (not necessarily different from p) of round r. q will eventually fast-propose V in the first instance for which it has not fast-proposed yet. The "`2a`" message it generates will eventually trigger the termination of the instance and have l learn V.

D TLA$^+$ Modules

D.1 M-Consensus

MODULE *MConsensus*

CONSTANTS *Proposer*, *Learner*, *Value*, *Nil*, *none*

INSTANCE *VMapping* WITH *Domain* $\leftarrow$ *Proposer*

VARIABLES *proposed*, *learned*

TypeInv asserts a type invariant; the assertion that *TypeInv* is always true is a property of (implied by) the specification

$$\begin{aligned} \mathit{TypeInv} \triangleq\ & \land\ \mathit{proposed} \subseteq \mathit{Value} \\ & \land\ \mathit{learned}\ \in [\mathit{Learner}\ \to \mathit{ValMap}] \end{aligned}$$

Init is the initial predicate.

$$\begin{aligned} \mathit{Init} \triangleq\ & \land\ \mathit{proposed} = \{\} \\ & \land\ \mathit{learned} = [l \in \mathit{Learner} \mapsto \mathit{Bottom}] \end{aligned}$$

$$\mathit{IsProposed}(m) \triangleq\ \forall\, p \in \mathit{Dom}(m) : m[p] \in (\mathit{proposed} \cup \{\mathit{Nil}\})$$

$$\mathit{IsTrivial}(m) \triangleq\ m = [p \in \mathit{Proposer} \mapsto \mathit{Nil}]$$

We now define the two actions of proposing a value and learning a mapping. The Learn action sets *learned*[*l*] to a mapping extending its present value.

$$\begin{aligned} \mathit{Propose} \triangleq\ & \land\ \exists\, v \in \mathit{Value} : \\ & \qquad \mathit{proposed}' = \mathit{proposed} \cup \{v\} \\ & \land\ \text{UNCHANGED}\ \langle \mathit{learned} \rangle \end{aligned}$$

$$\begin{aligned} \mathit{Learn}(l) \triangleq\ & \land\ \exists\, v \in \mathit{ValMap} : \\ & \qquad \land\ \mathit{IsProposed}(v) \\ & \qquad \land\ \forall\, l2 \in \mathit{Learner} : \mathit{AreCompatible}(v,\ \mathit{learned}[l2]) \\ & \qquad \land\ \neg \mathit{IsTrivial}(\mathit{LUB}(\{\mathit{learned}[r] : r \in \mathit{Learner}\}) \sqcup v) \\ & \qquad \land\ \mathit{learned}' = [\mathit{learned}\ \text{EXCEPT}\ ![l] = @ \sqcup v] \\ & \land\ \text{UNCHANGED}\ \langle \mathit{proposed} \rangle \end{aligned}$$

Next is the complete next-state action; *Spec* is the complete specification.

$$\mathit{Next} \triangleq\ \mathit{Propose} \lor \exists\, l \in \mathit{Learner} : \mathit{Learn}(l)$$

$$\mathit{Spec} \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{proposed},\, \mathit{learned} \rangle}$$

We now define the three safety properties as temporal formulas and assert that they and the type-correctness invariant are properties of the specification.

$$\begin{aligned}
\mathit{Nontriviality} &\triangleq \forall\, l \in \mathit{Learner} : \\
&\qquad \Box(\mathit{IsProposed}(\mathit{learned}[l]) \land \neg \mathit{IsTrivial}(\mathit{learned}[l]))
\end{aligned}$$

$$\begin{aligned}
\mathit{Stability} &\triangleq \forall\, l \in \mathit{Learner} : \\
&\qquad \Box(\exists\, v \in \mathit{ValMap} : \mathit{learned}[l] = v \Rightarrow \Box(v \sqsubseteq \mathit{learned}[l]))
\end{aligned}$$

$$\begin{aligned}
\mathit{Consistency} \triangleq\ & \Box(\ \land \mathit{IsCompatible}(\{\mathit{learned}[l] : l \in \mathit{Learner}\}) \\
& \qquad \land \neg \mathit{IsTrivial}(\mathit{LUB}(\{\mathit{learned}[r] : r \in \mathit{Learner}\})))
\end{aligned}$$

THEOREM $\mathit{Spec} \Rightarrow (\Box\, \mathit{TypeInv}) \land \mathit{Nontriviality} \land \mathit{Stability} \land \mathit{Consistency}$

D.2 Atomic Broadcast

MODULE *ABcast*

EXTENDS *Sequences*, *Naturals*

CONSTANTS *Proposer*, *Learner*, *Value*

VARIABLES *broadcast*, *delivered*

TypeInv asserts a type invariant; the assertion that *TypeInv* is always true is a property of (implied by) the specification

$$\begin{aligned}
\mathit{TypeInv} \triangleq\ & \land\ \mathit{broadcast} \subseteq \mathit{Value} \\
& \land\ \mathit{delivered} \in [\mathit{Learner} \to \mathit{Seq}(\mathit{Value})]
\end{aligned}$$

Init is the initial predicate.

$$\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{broadcast} = \{\} \\
& \land\ \mathit{delivered} = [l \in \mathit{Learner} \mapsto \langle\rangle]
\end{aligned}$$

$$\begin{aligned}
& s1 \sqsubseteq s2 \triangleq \\
& \quad \land\ \mathit{Len}(s1) \leq \mathit{Len}(s2) \\
& \quad \land\ \forall\, i \in 1 \,..\, \mathit{Len}(s1) : s1[i] = s2[i]
\end{aligned}$$

We now define the two actions of broadcasting a value and learning a sequence. The Learn action sets *learned*[*l*] to a sequence extending its present value.

$$\begin{aligned}
\mathit{Broadcast} \triangleq\ & \land\ \exists\, m \in \mathit{Value} : \\
& \qquad \mathit{broadcast}' = \mathit{broadcast} \cup \{m\} \\
& \land\ \text{UNCHANGED } \langle \mathit{delivered} \rangle
\end{aligned}$$

$$\mathit{Learn}(l) \triangleq\ \land\ \exists\, v \in \mathit{broadcast} :$$

$$
\begin{array}{l}
\quad \land \neg \exists\, i \in 1 \,..\, Len(delivered[l]) : \\
\qquad delivered[l][i] = v \\
\quad \land \forall\, l2 \in Learner : \\
\qquad \lor\ delivered[l] \circ \langle v \rangle \sqsubseteq delivered[l2] \\
\qquad \lor\ delivered[l2] \sqsubseteq delivered[l] \\
\quad \land\ delivered' = [delivered \text{ EXCEPT } ![l] = @ \circ \langle v \rangle] \\
\land \text{ UNCHANGED } \langle broadcast \rangle
\end{array}
$$

Next is the complete next-state action; *Spec* is the complete specification.

$$Next \triangleq Broadcast \lor \exists\, l \in Learner : Learn(l)$$

$$Spec \triangleq Init \land \Box[Next]_{\langle broadcast,\, delivered \rangle}$$

We now define the three safety properties as temporal formulas and assert that they and the type-correctness invariant are properties of the specification.

$$
\begin{array}{l}
Nontriviality \triangleq \forall\, l \in Learner : \\
\qquad \Box(delivered[l] \in Seq(broadcast))
\end{array}
$$

$$
\begin{array}{l}
Stability \triangleq \forall\, l \in Learner : \\
\qquad \Box(\exists\, s \in Seq(Value) : delivered[l] = s \\
\qquad\quad \Rightarrow \Box(s \sqsubseteq delivered[l]))
\end{array}
$$

$$
\begin{array}{l}
Consistency \triangleq \forall\, l1,\, l2 \in Learner : \\
\qquad \Box(\ \lor\ delivered[l1] \sqsubseteq delivered[l2] \\
\qquad\quad \lor\ delivered[l2] \sqsubseteq delivered[l1])
\end{array}
$$

THEOREM $Spec \Rightarrow (\Box\, TypeInv) \land Nontriviality \land Stability \land Consistency$

D.3 Value Mappings

MODULE *VMapping*

This module defines constants and operators for dealing with value mappings.

LOCAL INSTANCE *FiniteSets* The *FiniteSets* module defines the operation *Len*

We declare the sets *Domain* and *Value* as parameters.

CONSTANTS *Domain*, *Value*, *Nil*, *none*

Nil is a no-value used to map an element in *Domain* to nothing.

$Nil \triangleq$ CHOOSE $n : n \notin Value$

ASSUME $Nil \notin Value$

ValMap defines the set of all valid value mappings. It is composed of any function that maps a subset of the domain to values or *Nil*.

$$ValMap \triangleq \text{UNION } \{[PS \to Value \cup \{Nil\}] : PS \in \text{SUBSET } Domain\}$$

none is defined to be something that is not a *ValMap*

$$none \triangleq \text{CHOOSE } n : n \notin ValMap$$

$$\text{ASSUME } none \notin ValMap$$

$$\text{ASSUME } none \neq Nil$$

$$\text{ASSUME } none \notin Value$$

We define *Bottom* to be the "empty" *ValMap*, that is, a *ValMap* function whose domain is the empty set. In TLA, a function with empty domain is defined to be equal to the empty sequence, which allows the simplification below. We use *Bottom* instead of $\perp$.

$$Bottom \triangleq \langle\rangle$$

For simplicity, $Dom(f)$ is defined to be the domain of function f.

$$Dom(f) \triangleq \text{DOMAIN } f$$

A *SingleMap* maps a single domain element to a *Value* or *Nil*.

$$SingleMap \triangleq [p : Domain,\ v : Value \cup \{Nil\}]$$

$SM(p, v)$ defines a *SingleMap* from domain element p to value v.

$$SM(p,\ v) \triangleq [p \mapsto p,\ v \mapsto v]$$

The basic operation over a *ValMap* is $vm \bullet sm$. It agregates a *SingleMap* to a *ValMap*. It is well-defined only if vm is a *ValueMap* and sm is a *SingleMap*.

$$vm \bullet sm \triangleq [p \in Dom(vm) \cup \{sm.p\} \mapsto \text{IF } p \in Dom(vm) \text{ THEN } vm[p] \text{ ELSE } sm.v]$$

ValMap is a c-struct and admits all existing operators for c-structs. We define them in the following specifically for the *ValMap* type, which allows many simplifications and optimizations.

A *ValMap* v is a prefix of a *ValMap* w ($v \sqsubseteq w$) if it can be extended to w by a sequence of $\bullet$ applications with single mappings. This can be verified in a simplified way by checking if the domain of v is a subset of the domain of w and for every element in the domain of v, its mapped value in v is equal to its mapped value in w. If $v \sqsubseteq w$, we say that v is a prefix of w or that w extends v. We extend the definition of $v \sqsubseteq w$ so that it is true if both v and w equals none.

$$v \sqsubseteq w \triangleq \lor \land v \neq none \land w \neq none \land Dom(v) \subseteq Dom(w)$$

$$\begin{array}{ll} & \land \forall\, e \in Dom(v) : v[e] = w[e] \\ \lor & \land\ v = none \\ & \land\ w = none \end{array}$$

A *ValMap* v is a strict prefix of a *Valmap* w ($v \sqsubset w$) if it is a prefix of w and it is different from w.

$$v \sqsubset w \triangleq (v \sqsubseteq w) \land (v \neq w)$$

$GLB(T)$ is the greatest lower bound of a set of value mappings. It is a *ValMap* u that is a prefix to all *ValMaps* in T but is not a prefix of any other *ValMap* that is also a prefix of all *ValMaps* in T. It is more simply defined as a function that maps each element that belongs to the domain intersection of all mappings and whose mapped value in all mappings is the same to its mapped value in all value mappings.

$$\begin{array}{lll} GLB(T) \triangleq & \text{LET } witness & \triangleq \text{ CHOOSE } f \in T : \text{TRUE} \\ & CInter & \triangleq \{p \in Dom(witness) : \\ & & \quad \forall f \in T : \land p \in Dom(f) \\ & & \qquad\qquad \land f[p] = witness[p]\} \\ & \text{IN} & [p \in CInter \mapsto witness[p]] \end{array}$$

$v \sqcap w$ is defined to be the greatest lower bound for the set $\{v, w\}$

$$v \sqcap w \triangleq GLB(\{v, w\})$$

A *ValMap* v is defined to be compatible with a *ValMap* w if they are both prefixes of a *ValMap* u. It is equivalent to verifying if their common domain elements are mapped to the same values.

$$AreCompatible(v, w) \triangleq \forall\, e \in Dom(v) \cap Dom(w) : v[e] = w[e]$$

A set of *ValMaps* is compatible if its elements are pairwise compatible.

$$IsCompatible(S) \triangleq \forall\, v, w \in S : AreCompatible(v, w)$$

$LUB(T)$ is the least upper bound of a set of value mappings. It is a *ValMap* u that extends all *ValMaps* in T but does not extend any other *ValMap* that also extends all *ValMaps* in T. It is more simply defined as a function that maps each element that belongs to the domain of any of the mappings to its mapped value on any of the mappings whose domain it belongs to. It is only well-defined if T is a set of compatible value mappings.

$$\begin{array}{ll} LUB(T) \triangleq & [p \in \text{UNION } \{Dom(f) : f \in T\} \mapsto \\ & \quad (\text{CHOOSE } f \in T : p \in Dom(f))[p]] \end{array}$$

$v \sqcup w$ is defined to be the least upper bound for the set $\{v, w\}$

$$v \sqcup w \triangleq LUB(\{v, w\})$$

D.4 Order Relations

MODULE *OrderRelations*

We make some definitions for an arbitrary ordering relation $\sqsubseteq$ on a set S. The module will be used by *instantiang* $\sqsubseteq$ and S with a particular operator and Set.

CONSTANTS S, $\_ \sqsubseteq \_$

We define *IsPartialOrder* to be the assertion that $\sqsubseteq$ is an (irreflexive) partial order on a set S, and *IsTotalOrder* to be the assertion that it is a total ordering of S.

$$\begin{aligned}
&\mathit{IsPartialOrder} \triangleq \\
&\quad \land \forall\, u,\, v,\, w \in S : (u \sqsubseteq v) \land (v \sqsubseteq w) \Rightarrow (u \sqsubseteq w) \\
&\quad \land \forall\, u,\, v \in S \quad : (u \sqsubseteq v) \land (v \sqsubseteq u) \Rightarrow (u = v)
\end{aligned}$$

$$\begin{aligned}
&\mathit{IsTotalOrder} \triangleq \\
&\quad \land \mathit{IsPartialOrder} \\
&\quad \land \forall\, u,\, v \in S : (u \sqsubseteq v) \lor (v \sqsubseteq u)
\end{aligned}$$

We now define the glb (greatest lower bound) and lub (least upper bound) operators. To define *GLB*, we first define $\mathit{IsLB}(lb,\, T)$ to be true iff lb is a lower bound of T, and $\mathit{IsGLB}(lb,\, T)$ to be true iff lb is a glb of T. the value of $\mathit{GLB}(T)$ is unspecified if T has no glb. The definition for upper bounds are analogous.

$$\begin{aligned}
&\mathit{IsLB}(lb,\, T) \triangleq \land\, lb \in S \\
&\qquad\qquad\qquad\quad \land \forall\, v \in T : lb \sqsubseteq v \\
&\mathit{IsGLB}(lb,\, T) \triangleq \land\, \mathit{IsLB}(lb,\, T) \\
&\qquad\qquad\qquad\quad \land \forall\, v \in S : \mathit{IsLB}(v,\, T) \Rightarrow (v \sqsubseteq lb) \\
&\mathit{GLB}(T) \triangleq \text{CHOOSE } lb \in S : \mathit{IsGLB}(lb,\, T) \\
&v \sqcap w \triangleq \mathit{GLB}(\{v,\, w\}) \\
&\mathit{IsUB}(ub,\, T) \triangleq \land\, ub \in S \\
&\qquad\qquad\qquad\quad \land \forall\, v \in T : v \sqsubseteq ub \\
&\mathit{IsLUB}(ub,\, T) \triangleq \land\, \mathit{IsUB}(ub,\, T) \\
&\qquad\qquad\qquad\quad \land \forall\, v \in S : \mathit{IsUB}(v,\, T) \Rightarrow (ub \sqsubseteq v) \\
&\mathit{LUB}(T) \triangleq \text{CHOOSE } ub \in S : \mathit{IsLUB}(ub,\, T) \\
&v \sqcup w \triangleq \mathit{LUB}(\{v,\, w\})
\end{aligned}$$

D.5 Paxos Constants

MODULE *PaxosConstants*

This module defines the parameters and data structures for our algorithms.

EXTENDS *FiniteSets*

RNum is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called *Zero*.

CONSTANTS *RNum*, $\_ \preceq \_$, *Zero*

We assume $\preceq$ is a total ordering of the set *RNum* of round numbers.

ASSUME LET $PO \triangleq$ INSTANCE *OrderRelations* WITH $S \leftarrow RNum$, $\sqsubseteq \leftarrow \preceq$
IN *PO*!*IsTotalOrder*

We define $i \prec j$ to be true iff $i \preceq j$ for two different rounds i and j.

$i \prec j \triangleq (i \preceq j) \land (i \neq j)$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq$ CHOOSE $i \in B : \forall\, j \in B : j \preceq i$

Are parameters of this module:
- A set *Proposer* of proposer agents,
- A set *Learner* of learner agents,
- A set *Acceptor* of acceptor agents,
- An operator *Quorum* that returns the acceptor quorums of a round,
- An operator *CfProposer* that returns the collision-fast proposers of a round,
- And a set *Value* of proposable Values.

CONSTANTS *Proposer*, *Learner*, *Acceptor*, *Quorum*(_), *CfProposer*(_), *Value*

$Nil \triangleq$ CHOOSE $n : n \notin Value$
$none \triangleq$ CHOOSE $n : n \notin Value$

The problem of *MConsensus* is defined in terms of a value mapping set whose *Domain* is the set of proposers

INSTANCE *VMapping* WITH *Domain* $\leftarrow$ *Proposer*

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

$QuorumAssumption \triangleq$
$\forall\, i \in RNum :$
$\land\, Quorum(i) \subseteq$ SUBSET *Acceptor*
$\land\, \forall\, Q \in Quorum(i) : IsFiniteSet(Q)$
$\land\, \forall\, j \in RNum :$
$\forall\, Q \in Quorum(i),\, R \in Quorum(j) : Q \cap R \neq \{\}$

ASSUME *QuorumAssumption*

Over the set of collision-fast proposers for a round i, we only assume they are a finite subset of the proposers.

$$
\begin{array}{l}
CfProposerAssumption \triangleq \\
\quad \forall\, i \in RNum : \\
\qquad \land\ CfProposer(i) \in \text{SUBSET } Proposer \\
\qquad \land\ IsFiniteSet(CfProposer(i))
\end{array}
$$

ASSUME $CfProposerAssumption$

We say that a value mapping is valued iff at least one element of its domain is mapped to a value ($\neq Nil$). Our algorithm makes sure that acceptors will only accept valued mappings, so that we can guarantee *Nontriviality*.

$$IsValued(m) \triangleq \exists\, p \in Dom(m) : m[p] \neq Nil$$

We define the Nil-extension of a valued mapping v for a set P of proposers to be the *ValMap* resulting from adding to v the single mapping $p \to Nil$, for every proposer p in $P \setminus Dom(v)$.

$$
\begin{array}{l}
NilExtension(v,\ P) \triangleq \\
\quad \text{IF } v = none \text{ THEN } none \\
\qquad\qquad \text{ELSE } [p \in Dom(v) \cup P \mapsto \text{IF } p \in Dom(v) \text{ THEN } v[p] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } Nil]
\end{array}
$$

We define *BallotArray* to be the set of all ballot arrays. We represent a ballot array as a record, where we write $\beta_a[m]$ as $\beta.vote[a][m]$ and $\hat{\beta}_a$ as $\beta.rnd[a]$.

$$
\begin{array}{l}
BallotArray \triangleq \\
\quad \{beta \in [vote : [Acceptor \to [RNum \to ValMap \cup \{none\}]], \\
\qquad\qquad\ rnd\ : [Acceptor \to RNum]] : \\
\qquad \forall\, a \in Acceptor : \\
\qquad\quad \land\ IsFiniteSet(\{m \in RNum : beta.vote[a][m] \neq none\}) \\
\qquad\quad \land\ \forall\, m \in RNum : \\
\qquad\qquad\quad \land\ (beta.rnd[a] \prec m) \Rightarrow (beta.vote[a][m] = none) \\
\qquad\qquad\quad \land\ (beta.vote[a][m] \neq none) \Rightarrow IsValued(beta.vote[a][m])\}
\end{array}
$$

We define *CfPropArray* to be the set of all proposal arrays.

$$PropArray \triangleq [Proposer \to [RNum \to Value \cup \{Nil\} \cup \{none\}]]$$

We now formalize the definitions of *chosen at*, *safe at*, etc. We translate the *English* terms into obvious operator names. For example, $IsChosenAt(v,\ m,\ \beta,\ \gamma\ )$ is defined to be true iff v is chosen at m in $\langle\ \beta, \gamma\ \rangle$, assuming that v is a *ValMap*, m is a round number, β is a ballot array, and γ is a proposal array. (We don't care what $IsChosenAt(v,\ m,\ \beta,\ \gamma\ )$ means for other values of v, m, β, and γ.) We also assert the two propositions as theorems.

$$IsChosenAt(v,\ m,\ beta,\ gamma) \triangleq$$

$$
\begin{array}{l}
\quad \text{LET } NilP \triangleq \{p \in CfProposer(m) : gamma[p][m] = Nil\} \\
\quad \text{IN } \quad \exists\, Q \in Quorum(m) : \\
\qquad\quad \forall\, a \in Q : (v \sqsubseteq NilExtension(beta.vote[a][m],\, NilP))
\end{array}
$$

$$
IsChosenIn(v,\, beta,\, gamma) \triangleq \exists\, m \in RNum : IsChosenAt(v,\, m,\, beta,\, gamma)
$$

$$
\begin{array}{l}
IsChoosableAt(v,\, m,\, beta,\, gamma) \triangleq \\
\quad \exists\, Q \in Quorum(m) : \\
\quad\ \text{LET } P \triangleq \{p \in Proposer : gamma[p][m] \in \{Nil,\, none\}\} \\
\quad\ \text{IN } \quad \forall\, a \in Q : \\
\qquad\quad (m \prec beta.rnd[a]) \Rightarrow \\
\qquad\qquad\quad (v \sqsubseteq NilExtension(beta.vote[a][m],\, Proposer))
\end{array}
$$

$$
\begin{array}{l}
IsSafeAt(v,\, m,\, beta,\, gamma) \triangleq \\
\quad \forall\, k \in RNum : \\
\qquad (k \prec m) \Rightarrow \forall\, w \in ValMap : \\
\qquad\qquad\qquad IsChoosableAt(w,\, k,\, beta,\, gamma) \Rightarrow (w \sqsubseteq v)
\end{array}
$$

$$
\begin{array}{l}
IsSafe(beta,\, gamma) \triangleq \\
\quad \forall\, a \in Acceptor,\, k \in RNum : \\
\qquad (beta.vote[a][k] \neq none) \Rightarrow IsSafeAt(beta.vote[a][k],\, k,\, beta,\, gamma)
\end{array}
$$

$$
\begin{array}{l}
Proposition1 \triangleq \\
\quad \forall\, beta \in BallotArray,\, gamma \in PropArray : \\
\qquad IsSafe(beta,\, gamma) \Rightarrow \\
\qquad\qquad\quad IsCompatible(\{v \in ValMap : IsChosenIn(v,\, beta,\, gamma)\})
\end{array}
$$

THEOREM $Proposition1$

$$
\begin{array}{l}
IsConservative(beta,\, gamma) \triangleq \\
\quad \forall\, m \in RNum,\, a,\, b \in Acceptor : \\
\qquad \land\, beta.vote[a][m] \neq none \\
\qquad \land\, beta.vote[b][m] \neq none \\
\qquad \Rightarrow\ \land\, AreCompatible(beta.vote[a][m],\, beta.vote[b][m]) \\
\qquad\quad\ \ \land \forall\, p \in Dom(beta.vote[b][m]) \setminus Dom(beta.vote[a][m]) : \\
\qquad\qquad\quad beta.vote[b][m][p] = gamma[p][m]
\end{array}
$$

$$
\begin{array}{l}
ProvedSafe(Q,\, m,\, beta) \triangleq \\
\quad \text{IF } \forall\, a \in Q,\, i \in RNum : (i \prec m) \Rightarrow (beta.vote[a][i] = none) \\
\qquad \text{THEN } Bottom \\
\qquad \text{ELSE LET } k \triangleq Max(\{i \in RNum : \\
\qquad\qquad\qquad\qquad (i \prec m) \land (\exists\, a \in Q : beta.vote[a][i] \neq none)\}) \\
\qquad\qquad\qquad\quad AS \triangleq \{a \in Q : beta.vote[a][k] \neq none\}
\end{array}
$$

$$
\begin{array}{ll}
 & G \triangleq \{beta.vote[a][k] : a \in AS\} \\
\text{IN} & NilExtension(LUB(G),\ Proposer)
\end{array}
$$

$Proposition2 \triangleq$
$\forall\, m \in RNum,\ beta \in BallotArray,\ gamma \in PropArray :$
$\forall\, Q \in Quorum(m) :$
$\land\ IsSafe(beta,\ gamma)$
$\land\ IsConservative(beta,\ gamma)$
$\land \forall\, a \in Q : m \preceq beta.rnd[a]$
$\Rightarrow\ IsSafeAt(ProvedSafe(Q,\ m,\ beta),\ m,\ beta,\ gamma)$

THEOREM $Proposition2$

D.6 Abstract Collision-Fast Paxos

MODULE *AbstractCFPaxos*

Abstract algorithm

EXTENDS *PaxosConstants*

The algorithm's variables:
- proposed: set of proposed values
- learned: array that maps each learner to its currently learned *ValMap*
- *bA*: a ballot array that keeps current round and history of votes for each acceptor.
- *pA*: a proposal array that keeps the history of proposals for each proposer.
- *minTried*: a vector with the safe initial value to be accepted at each round.

VARIABLES *proposed*, *learned*, *bA*, *pA*, *minTried*

The type invariant asserts that the specification preserves the types of the variables according to the definition below.

$$
\begin{array}{ll}
TypeInv \triangleq & \land\ proposed \subseteq Value \\
 & \land\ learned \in [Learner \to ValMap] \\
 & \land\ bA \in BallotArray \\
 & \land\ pA \in PropArray \\
 & \land\ minTried \in [RNum \to ValMap \cup \{none\}]
\end{array}
$$

Initial state of the specification

$$
\begin{array}{ll}
Init \triangleq & \land\ proposed = \{\} \\
 & \land\ learned = [l \in Learner \mapsto Bottom] \\
 & \land\ bA = [vote \mapsto [a \in Acceptor \mapsto [m \in RNum \mapsto none]], \\
 & \qquad\quad rnd \mapsto [a \in Acceptor \mapsto Zero]] \\
 & \land\ pA = [p \in Proposer \mapsto [i \in RNum \mapsto none]]
\end{array}
$$

$\land\ minTried = [i \in RNum \mapsto \text{IF } i = Zero \text{ THEN } Bottom \text{ ELSE } none\,]$

Propose(V) adds value V to the set *proposed* if it is not there yet.

$Propose(V) \triangleq$
$\land\ V \notin proposed$
$\land\ proposed' = proposed \cup \{V\}$
$\land$ UNCHANGED $\langle learned,\ bA,\ pA,\ minTried\rangle$

JoinRound$(a,\ m)$ changes the current round of acceptor a to m.

$JoinRound(a,\ m) \triangleq$
$\land\ bA.rnd[a] \prec m$
$\land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m]$
$\land$ UNCHANGED $\langle proposed,\ learned,\ pA,\ minTried\rangle$

StartRound$(m,\ Q)$ sets *minTried*$[m]$ to a value safe at m in bA, according to the definition of *ProvedSafe*$(Q,\ m,\ bA)$.

$StartRound(m,\ Q) \triangleq$
$\land\ minTried[m] = none$
$\land\ \forall\, a \in Q : m \preceq bA.rnd[a]$
$\land\ minTried' = [minTried \text{ EXCEPT } ![m] = ProvedSafe(Q,\ m,\ bA)]$
$\land$ UNCHANGED $\langle proposed,\ learned,\ bA,\ pA\rangle$

Suggest$(p,\ m,\ V)$ changes $pA[p][m]$ from none to a value or *Nil* (this last, only if other proposer has suggested a value or *minTried*$[m][p]$ equals *Nil*).

$Suggest(p,\ m,\ V) \triangleq$
$\land\ \lor\ minTried[m] \notin \{Bottom,\ none\} \land V = minTried[m][p]$
$\quad\lor\ V \in proposed$
$\quad\lor\ \land\ V = Nil$
$\qquad\land\ \exists\, pv \in CfProposer(m) : pA[pv][m] \in Value$
$\land\ pA[p][m] = none$
$\land\ pA' = [pA \text{ EXCEPT } ![p][m] = V]$
$\land$ UNCHANGED $\langle proposed,\ learned,\ bA,\ minTried\rangle$

ClassicVote$(a,\ m,\ v)$ extends the vote of acceptor a for round m, changing it for value v if the conditions below are satisfied.

$ClassicVote(a,\ m,\ v) \triangleq$
$\land\ bA.rnd[a] \preceq m$
$\land\ IsValued(v)$
$\land\ minTried[m] \neq none$
$\land\ minTried[m] \sqsubseteq v$

$\land$ LET $sp(p) \triangleq SM(p,\ pA[p][m])$
$pS \triangleq \{p \in CfProposer(m) : pA[p][m] \neq none\}$
$maxTried \triangleq LUB(\{minTried[m] \bullet sp(p) : p \in pS\})$
IN $v \sqsubseteq NilExtension(maxTried,\ Proposer \setminus CfProposer(m))$
$\land \lor bA.vote[a][m] = none$
$\lor bA.vote[a][m] \sqsubset v$
$\land bA' = [bA$ EXCEPT $!.rnd[a] = m,\ !.vote[a][m] = v]$
$\land$ UNCHANGED $\langle proposed,\ learned,\ pA,\ minTried\rangle$

AbstractLearn$(l,\ v)$ extends $learned[l]$ with v iff v is chosen in $\langle bA,\ pA\rangle$.

$AbstractLearn(l,\ v) \triangleq$
$\land IsChosenIn(v,\ bA,\ pA)$
$\land learned' = [learned$ EXCEPT $![l] = learned[l] \sqcup v]$
$\land$ UNCHANGED $\langle proposed,\ bA,\ pA,\ minTried\rangle$

Next defines the next-state relation and *Spec* is the complete specification.

$Next \triangleq$
$\lor \exists\, V \in Value : Propose(V)$
$\lor \exists\, a \in Acceptor,\ m \in RNum : JoinRound(a,\ m)$
$\lor \exists\, m \in RNum :$
$\lor \exists\, Q \in Quorum(m) : StartRound(m,\ Q)$
$\lor \exists\, p \in CfProposer(m),\ V \in Value \cup \{Nil\} : Suggest(p,\ m,\ V)$
$\lor \exists\, a \in Acceptor,\ m \in RNum,\ v \in ValMap : ClassicVote(a,\ m,\ v)$
$\lor \exists\, l \in Learner,\ v \in ValMap : AbstractLearn(l,\ v)$

$Spec \triangleq Init \land \Box[Next]_{\langle proposed,\ learned,\ bA,\ pA,\ minTried\rangle}$

The theorems below asserts that the spec ensures the type invariant and implements the general specification of *MConsensus*.

THEOREM $Spec \Rightarrow \Box\, TypeInv$

$MC \triangleq$ INSTANCE *MConsensus*
THEOREM $Spec \Rightarrow MC!Spec$

D.7 Distributed Abstract Collision-Fast Paxos

MODULE *DistAbsCFPaxos*

EXTENDS *PaxosConstants*

The algorithm's variables are the same as in the abstract algorithm plus *msgs*: set of system messages. Since we are specifying only safety, message loss is simply implemented by not executing actions that depend on the message, without having to take it explicitly out of the set *msgs*. Moreover, duplicate messages are implemented by keeping messages in *msgs*, since they could possibly trigger the same action multiple times.

VARIABLES $proposed$, $learned$, bA, pA, $minTried$, $msgs$

Msg is the set containing all possible messages by the algorithm. For clarity, we use records instead of sequences to represent messages.

$$\begin{aligned} Msg \triangleq\ & [type : \{\text{“propose”}\},\ val : Value] \cup \\ & [type : \{\text{“1a”}\},\ rnd : RNum] \cup \\ & [type : \{\text{“1b”}\},\ rnd : RNum,\ acc : Acceptor, \\ & \qquad vote : [RNum \to ValMap \cup \{none\}]] \cup \\ & [type : \{\text{“2S”}\},\ rnd : RNum,\ val : ValMap] \cup \\ & [type : \{\text{“2b”}\},\ rnd : RNum,\ acc : Acceptor,\ val : ValMap] \cup \\ & [type : \{\text{“2a”}\},\ rnd : RNum,\ val : SingleMap] \end{aligned}$$

Type Invariant

$$\begin{aligned} TypeInv \triangleq\ & \land\ proposed \subseteq Value \\ & \land\ learned \in [Learner \to ValMap] \\ & \land\ bA \in BallotArray \\ & \land\ pA \in PropArray \\ & \land\ minTried \in [RNum \to ValMap \cup \{none\}] \\ & \land\ msgs \subseteq Msg \end{aligned}$$

Initial state

$$\begin{aligned} Init \triangleq\ & \land\ proposed = \{\} \\ & \land\ learned = [l \in Learner \mapsto Bottom] \\ & \land\ bA = [vote \mapsto [a \in Acceptor \mapsto [m \in RNum \mapsto none]], \\ & \qquad\ rnd\ \mapsto [a \in Acceptor \mapsto Zero]] \\ & \land\ pA = [p\ \in Proposer \mapsto [i \in RNum \mapsto none]] \\ & \land\ minTried = [i \in RNum \mapsto \text{IF}\ i = Zero\ \text{THEN}\ Bottom \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ none] \\ & \land\ msgs = \{[type \mapsto \text{“2S”},\ rnd \mapsto Zero,\ val \mapsto Bottom]\} \end{aligned}$$

Actions

Action $Send(msg)$ implements the sending of message *msg*.

$$Send(msg) \triangleq msgs' = msgs \cup \{msg\}$$

Propose(V) executes a value proposal. In the specification we make no distinction between a proposal made by a collision-fast proposer and one made by an ordinary external proposer. The difference lies on whether the "propose" message will be local to a processor or not.

$$
\begin{array}{l}
Propose(V) \triangleq \\
\quad \land\ V \notin proposed \\
\quad \land\ proposed' = proposed \cup \{ V \} \\
\quad \land\ Send([type \mapsto \text{“propose”},\ val \mapsto V]) \\
\quad \land\ \text{UNCHANGED}\ \langle learned,\ bA,\ pA,\ minTried \rangle
\end{array}
$$

Action *Phase*1$a(m)$ triggers the start of a new round m. It sends a phase "1a" message for round m.

$$
\begin{array}{l}
Phase1a(m) \triangleq \\
\quad \land\ minTried[m] = none \\
\quad \land\ Send([type \mapsto \text{“1a”},\ rnd \mapsto m]) \\
\quad \land\ \text{UNCHANGED}\ \langle proposed,\ learned,\ bA,\ pA,\ minTried \rangle
\end{array}
$$

Action *Phase*1$b(a,\ m)$ is executed by acceptor a in response to a phase "1a" message. The action is executed only once per round. It changes the current round of acceptor a to m and sends a phase "1b" message containing the current voting situation of a.

$$
\begin{array}{l}
Phase1b(a,\ m) \triangleq \\
\quad \land\ [type \mapsto \text{“1a”},\ rnd \mapsto m] \in msgs \\
\quad \land\ bA.rnd[a] \prec m \\
\quad \land\ bA' = [bA\ \text{EXCEPT}\ !.rnd[a] = m] \\
\quad \land\ Send([type \mapsto \text{“1b”},\ rnd \mapsto m,\ acc \mapsto a,\ vote \mapsto bA.vote[a]]) \\
\quad \land\ \text{UNCHANGED}\ \langle proposed,\ learned,\ pA,\ minTried \rangle
\end{array}
$$

Action *Phase*2*Start*(m) "starts" round m. It is enabled iff the round has not been started and a quorum of acceptors has sent phase "1b" messages for round m. It uses these "1b" messages to pick up a safe *ValMap* for round m, using *ProvedSafe*$(Q,\ m,\ beta)$ as defined in module *PaxosConstants*. *minTried*$[m]$ is set to this safe value and a phase "2S" message is sent to inform it.

$$
\begin{array}{l}
Phase2Start(m) \triangleq \\
\quad \land\ minTried[m] = none \\
\quad \land\ \exists\, Q \in Quorum(m) : \\
\qquad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.rnd = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.acc\ = a \\
\qquad \land\ \text{LET}\ 1bMsg \triangleq [a \in Q \mapsto \text{CHOOSE}\ msg \in msgs : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.rnd = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.acc\ = a]
\end{array}
$$

$$
\begin{array}{lll}
& beta & \triangleq [vote \mapsto [a \in Q \mapsto 1bMsg[a].vote], \\
& & \quad rnd \mapsto [a \in Q \mapsto m]] \\
& v & \triangleq ProvedSafe(Q,\ m,\ beta) \\
\text{IN} & \land\ minTried' = [minTried \text{ EXCEPT } ![m] = v] \\
& \land\ Send([type \mapsto \text{“2S”},\ rnd \mapsto m,\ val \mapsto v])
\end{array}
$$
$$
\land \text{ UNCHANGED } \langle proposed,\ learned,\ bA,\ pA \rangle
$$

Action $Phase2Prepare(p,\ m)$ is executed by proposer p, for round m. It is enabled iff $pA[p][m]$ is different from none and p has received a “2S” message containing a v-mapping different from $Bottom$. The action sets $pA[p][m]$ to $v[p]$.

$$
\begin{array}{l}
Phase2Prepare(p,\ m) \triangleq \\
\quad \land\ pA[p][m] = none \\
\quad \land\ \exists\, v \in ValMap : \\
\qquad \land\ [type \mapsto \text{“2S”},\ rnd \mapsto m,\ val \mapsto v] \in msgs \\
\qquad \land\ v \neq Bottom \\
\qquad \land\ pA' = [pA \text{ EXCEPT } ![p][m] = v[p]] \\
\quad \land \text{ UNCHANGED } \langle proposed,\ learned,\ bA,\ minTried,\ msgs \rangle
\end{array}
$$

Action $Phase2a(p,\ m,\ V)$ is executed by proposer p, for round m and value V. It is enabled iff p is a collision-fast proposer for m, it has received a phase “2S” message containing $Bottom$ and either a “propose” or a “2a” message for m, and $pA[p][m]$ equals none. The action sets $pA[p][m]$ to V if p received a $\langle$“propose”, $V\rangle$ message or to Nil otherwise (p received a phase “2a” message from another proposer). It also sends a phase “2a” message for round m with a single mapping from p to V.

$$
\begin{array}{l}
Phase2a(p,\ m,\ V) \triangleq \\
\quad \land\ p \in CfProposer(m) \\
\quad \land\ pA[p][m] = none \\
\quad \land\ [type \mapsto \text{“2S”},\ rnd \mapsto m,\ val \mapsto Bottom] \in msgs \\
\quad \land\ \lor\ [type \mapsto \text{“propose”},\ val \quad\ \mapsto V] \in msgs \\
\qquad \lor\ \land\ V = Nil \\
\qquad\quad \land\ \exists\, q \in CfProposer(m),\ U \in Value : \\
\qquad\qquad [type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto SM(q,\ U)] \in msgs \\
\quad \land\ pA' = [pA \text{ EXCEPT } ![p][m] = V] \\
\quad \land\ Send([type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto SM(p,\ V)]) \\
\quad \land \text{ UNCHANGED } \langle proposed,\ learned,\ bA,\ minTried \rangle
\end{array}
$$

Action $Phase2b(a,\ m,\ v)$ is executed by acceptor a, for round m and $ValMap$ v. It is enabled only if m is higher than or equal to the current round of a, either v is valued and came on a phase “2S” message or v is built out of a phase “2a” message whose value is a mapping from a proposer to a (non-Nil) value. Moreover, the current vote of a for m must be either none or a prefix of v. The action sets the current round of a to m and a's vote at m to v.

$$
\begin{array}{l}
Phase2b(a,\ m,\ v) \triangleq \\
\ \land\ bA.rnd[a] \preceq m \\
\ \land\ \lor\ \land\ [type \mapsto \text{“2S”},\ rnd \mapsto m,\ val \mapsto v] \in msgs \\
\qquad\ \land\ IsValued(v) \\
\qquad\ \land\ bA.vote[a][m] = none \\
\qquad \lor \exists\, s \in SingleMap : \\
\qquad\quad\ \land\ [type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto s] \in msgs \\
\qquad\quad\ \land\ s.v \neq Nil \\
\qquad\quad\ \land\ \lor\ \land\ bA.vote[a][m] = none \\
\qquad\qquad\qquad\ \land\ v = NilExtension(Bottom \bullet s,\ Proposer \setminus CfProposer(m)) \\
\qquad\qquad\ \lor\ \land\ bA.vote[a][m] \neq none \\
\qquad\qquad\qquad\ \land\ v = bA.vote[a][m] \bullet s \\
\ \land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m,\ !.vote[a][m] = v] \\
\ \land\ Send([type \mapsto \text{“2b”},\ rnd \mapsto m,\ acc \mapsto a,\ val \mapsto v]) \\
\ \land\ \text{UNCHANGED } \langle proposed,\ learned,\ pA,\ minTried \rangle
\end{array}
$$

Action *Learn* is executed by learner l, for a *ValMap* v. Let P be the set of proposers from which l has received phase “2a” messages with single mappings from them to *Nil*. The action is enabled if there is a quorum Q of acceptors from which l has received phase “2b” messages such that v is a prefix of the of the values in each of these messages Nil-extended for P. The action sets $learned[l]$ to the lub between its previous value and v.

$$
\begin{array}{l}
Learn(l,\ v) \triangleq \\
\ \land \exists\, m \in RNum : \\
\qquad \exists\, Q \in Quorum(m),\ P \in \text{SUBSET } CfProposer(m) : \\
\qquad\ \land \forall\, p \in P : [type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto SM(p,\ Nil)] \in msgs \\
\qquad\ \land \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type\ = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.rnd = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.acc\ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ v \sqsubseteq NilExtension(msg.val,\ P) \\
\ \land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
\ \land\ \text{UNCHANGED } \langle proposed,\ bA,\ pA,\ minTried,\ msgs \rangle
\end{array}
$$

Next defines the next-state relation and *Spec* is the complete specification.

$$
\begin{array}{lll}
Next \triangleq & \lor \exists\, V \in Value : Propose(V) \\
& \lor \exists\, m \in RNum : Phase1a(m) \\
& \lor \exists\, a \in Acceptor,\ m \in RNum : Phase1b(a,\ m) \\
& \lor \exists\, m \in RNum : Phase2Start(m) \\
& \lor \exists\, p \in Proposer,\ m \in RNum,\ V \in Value \cup \{Nil\} : \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Phase2a(p,\ m,\ V) \\
& \lor \exists\, a \in Acceptor,\ m \in RNum,\ v \in ValMap : Phase2b(a,\ m,\ v)
\end{array}
$$

$$\lor \exists\, l \in Learner,\, v \in ValMap : Learn(l,\, v)$$

$Spec \triangleq Init \land \Box[Next]_{\langle proposed,\, learned,\, bA,\, pA,\, minTried,\, msgs\rangle}$

The theorems below asserts that the spec ensures the type invariant and implements the general specification of *MConsensus*.

THEOREM $Spec \Rightarrow \Box\, TypeInv$

$MC \triangleq$ INSTANCE $MConsensus$

THEOREM $Spec \Rightarrow MC!Spec$

D.8 Collision-Fast Paxos

MODULE *DistCFPaxosLiv*

EXTENDS *FiniteSets* Standard module with basic operations for finite sets.

RNum is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called *Zero*.

CONSTANTS $RNum,\ \_ \preceq \_,\ Zero$

We assume $\preceq$ is a total ordering of the set *RNum* of round numbers. Module *OrderRelations* can be found in our complete technical report.

ASSUME LET $PO \triangleq$ INSTANCE *OrderRelations* WITH $S \leftarrow RNum,\ \sqsubseteq \leftarrow \preceq$

IN $PO!IsTotalOrder$

We define $i \prec j$ to be true iff $i \preceq j$ for two different rounds i and j.

$i \prec j \triangleq (i \preceq j) \land (i \neq j)$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq$ CHOOSE $i \in B : \forall\, j \in B : j \preceq i$

Are parameters of this module:
- A set *Proposer* of proposer agents,
- A set *Learner* of learner agents,
- A set *Coord* of coordinator agents,
- An operator *CoordOf* that returns the coordinator of a round,
- A set *Acceptor* of acceptor agents,
- An operator *Quorum* that returns the acceptor quorums of a round,
- An operator *CfProposer* that returns the collision-fast proposers of a round,
- A set *Value* of proposable Values,
- A special value *Nil* not in *Value*,

- And a special value none not in *Value*.

CONSTANTS $Proposer$, $Learner$, $Coord$, $CoordOf(\_)$, $Acceptor$,
$Quorum(\_)$, $CfProposer(\_)$, $Value$, Nil, $none$

ASSUME $Nil \notin Value$
ASSUME $none \notin Value$

The problem of *MConsensus* is defined in terms of a value mapping set whose *Domain* is the set of proposers

INSTANCE $VMapping$ WITH $Domain \leftarrow Proposer$

We make the assumption that, for every round r, r has a single coordinator responsible for it and every coordinator is is responsible for a round higher-numbered than r.

$$CoordAssumption \triangleq$$
$$\forall\, r \in RNum : \land\, CoordOf(r) \in Coord$$
$$\land\, \forall\, c \in Coord : \exists\, r2 \in RNum : \land\, r \prec r2$$
$$\land\, c = CoordOf(r2)$$

ASSUME $CoordAssumption$

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

$$QuorumAssumption \triangleq$$
$$\forall\, i \in RNum :$$
$$\land\, Quorum(i) \subseteq \text{SUBSET } Acceptor$$
$$\land\, \forall\, Q \in Quorum(i) : IsFiniteSet(Q)$$
$$\land\, \forall\, j \in RNum :$$
$$\forall\, Q \in Quorum(i),\, R \in Quorum(j) : Q \cap R \neq \{\}$$

ASSUME $QuorumAssumption$

Over the set of collision-fast proposers for a round i, we only assume they are a finite subset of the proposers.

$$CfProposerAssumption \triangleq$$
$$\forall\, i \in RNum :$$
$$\land\, CfProposer(i) \in \text{SUBSET } Proposer$$
$$\land\, IsFiniteSet(CfProposer(i))$$

ASSUME $CfProposerAssumption$

We say that a value mapping is valued iff at least one element of its domain is mapped to a value ($\neq Nil$). Our algorithm makes sure that acceptors will only accept valued mappings, so that we can guarantee *Nontriviality*.

$$IsValued(m) \triangleq \exists\, p \in Dom(m) : m[p] \neq Nil$$

We define the Nil-extension of a valued mapping v for a set P of proposers to be the *ValMap* resulting from adding to v the single mapping $p \to Nil$, for every proposer p in $P \setminus Dom(v)$.

$$\begin{array}{l} NilExtension(v,\ P) \triangleq \\ \quad \text{IF } v = none \text{ THEN } none \\ \qquad\qquad \text{ELSE } [p \in Dom(v) \cup P \mapsto \text{IF } p \in Dom(v) \text{ THEN } v[p] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } Nil] \end{array}$$

The algorithm's variables are the following:
- proposed: set of proposed values
- learned: array mapping each learner to its currently learned *ValMap*
- *rnd*: array mapping each acceptor to its current round.
- *vrnd*: array mapping each acceptor to the last round at which it accepted something.
- *vval*: array mapping each acceptor a to the *ValMap* accepted in $vrnd[a]$.
- *prnd*: array mapping each proposer to its current round.
- *pval*: array mapping each proposer p to the value it fast-proposed at round $prnd[p]$.
- *crnd*: array mapping each coordinator to its current round.
- *cval*: array mapping each coordinator to the initial *ValMap* it has picked for round $crnd[p]$.
- *msgs*: set of messages that implements the message passing subsystem
- *noncrashed*: set of non-crashed agents in the system.
- *amLeader*: array mapping each coordinator to TRUE or FALSE depending on whether it believes to be the leader or not
- *activep*: array mapping each coordinator to the set of proposers it currently believes to be active.

VARIABLES $proposed,\ learned,\ rnd,\ vrnd,\ vval,\ prnd,\ pval,\ crnd,\ cval,$
$msgs,\ noncrashed,\ amLeader,\ activep$

$$\begin{array}{l} aVars \triangleq \langle rnd,\ vrnd,\ vval \rangle \\ pVars \triangleq \langle prnd,\ pval \rangle \\ cVars \triangleq \langle crnd,\ cval \rangle \\ oVars \triangleq \langle proposed,\ learned,\ noncrashed,\ amLeader,\ activep \rangle \end{array}$$

Msg is the set containing all possible messages by the algorithm. For clarity, we use records instead of sequences to represent messages.

$$\begin{array}{ll} Msg \triangleq & [type : \{\text{“propose”}\},\ val : Value] \cup \\ & [type : \{\text{“1a”}\},\ rnd : RNum] \cup \\ & [type : \{\text{“1b”}\},\ rnd : RNum,\ acc : Acceptor, \\ & \qquad\qquad vrnd : RNum,\ vval : ValMap \cup \{none\}] \cup \\ & [type : \{\text{“2S”}\},\ rnd : RNum,\ val : ValMap] \ \cup \\ & [type : \{\text{“2b”}\},\ rnd : RNum,\ acc : Acceptor,\ val : ValMap] \cup \\ & [type : \{\text{“2a”}\},\ rnd : RNum,\ val : SingleMap] \end{array}$$

Type Invariant

$$
\begin{array}{l}
\mathit{TypeInv} \triangleq \\
\quad \land\ \mathit{proposed} \subseteq \mathit{Value} \\
\quad \land\ \mathit{learned} \in [\mathit{Learner} \to \mathit{ValMap}] \\
\quad \land\ \mathit{rnd} \in [\mathit{Acceptor} \to \mathit{RNum}] \\
\quad \land\ \mathit{vrnd} \in [\mathit{Acceptor} \to \mathit{RNum}] \\
\quad \land\ \mathit{vval} \in [\mathit{Acceptor} \to \mathit{ValMap} \cup \{\mathit{none}\}] \\
\quad \land\ \mathit{prnd} \in [\mathit{Proposer} \to \mathit{RNum}] \\
\quad \land\ \mathit{pval} \in [\mathit{Proposer} \to \mathit{Value} \cup \{\mathit{Nil}\} \cup \{\mathit{none}\}] \\
\quad \land\ \mathit{crnd} \in [\mathit{Coord} \to \mathit{RNum}] \\
\quad \land\ \mathit{cval} \in [\mathit{Coord} \to \mathit{ValMap} \cup \{\mathit{none}\}] \\
\quad \land\ \mathit{msgs} \subseteq \mathit{Msg} \\
\quad \land\ \mathit{noncrashed} \subseteq \mathit{Acceptor} \cup \mathit{Coord} \cup \mathit{Proposer} \cup \mathit{Learner} \\
\quad \land\ \mathit{amLeader} \in [\mathit{Coord} \to \text{BOOLEAN }] \\
\quad \land\ \mathit{activep} \in [\mathit{Coord} \to \text{SUBSET } \mathit{Proposer}]
\end{array}
$$

Initial state

$$
\begin{array}{lll}
\mathit{Init} \triangleq & \land\ \mathit{proposed} = \{\} \\
& \land\ \mathit{learned} = [l \in \mathit{Learner} \mapsto \mathit{Bottom}] \\
& \land\ \mathit{rnd} = [a \in \mathit{Acceptor} \mapsto \mathit{Zero}] \\
& \land\ \mathit{vrnd} = [a \in \mathit{Acceptor} \mapsto \mathit{Zero}] \\
& \land\ \mathit{vval} = [a \in \mathit{Acceptor} \mapsto \mathit{none}] \\
& \land\ \mathit{prnd} = [p \in \mathit{Proposer} \mapsto \mathit{Zero}] \\
& \land\ \mathit{pval} = [p \in \mathit{Proposer} \mapsto \mathit{none}] \\
& \land\ \mathit{crnd} = [c \in \mathit{Coord} \mapsto \mathit{Zero}] \\
& \land\ \mathit{cval} = [c \in \mathit{Coord} \mapsto \text{IF } c = \mathit{CoordOf}(\mathit{Zero}) \\
& \qquad\qquad\qquad\qquad\quad \text{THEN } \mathit{Bottom} \\
& \qquad\qquad\qquad\qquad\quad \text{ELSE } \mathit{none}] \\
& \land\ \mathit{msgs} = \{\} \\
& \land\ \mathit{noncrashed} = \mathit{Acceptor} \cup \mathit{Coord} \cup \mathit{Proposer} \cup \mathit{Learner} \\
& \land\ \mathit{amLeader} = [c \in \mathit{Coord} \mapsto \text{IF } c = \mathit{CoordOf}(\mathit{Zero}) \\
& \qquad\qquad\qquad\qquad\quad \text{THEN TRUE} \\
& \qquad\qquad\qquad\qquad\quad \text{ELSE FALSE}] \\
& \land\ \mathit{activep} = [c \in \mathit{Coord} \mapsto \mathit{Proposer}]
\end{array}
$$

Agent Actions

Action $\mathit{Send}(\mathit{msg})$ implements the sending of message msg.

$$\mathit{Send}(\mathit{msg}) \triangleq \mathit{msgs}' = \mathit{msgs} \cup \{\mathit{msg}\}$$

Propose(V) executes a value proposal. In the specification we make no distinction between a proposal made by a collision-fast proposer and one made by an ordinary external proposer. The difference lies on whether the "propose" message will be local to a processor or not.

$$
\begin{array}{l}
\mathit{Propose}(V) \triangleq \\
\quad \land\ V \notin \mathit{proposed} \\
\quad \land\ \mathit{proposed}' = \mathit{proposed} \cup \{ V \} \\
\quad \land\ \mathit{Send}([\mathit{type} \mapsto \text{“propose”},\ \mathit{val} \mapsto V]) \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{aVars},\ \mathit{pVars},\ \mathit{cVars},\ \mathit{learned},\ \mathit{noncrashed},\ \mathit{amLeader},\ \mathit{activep} \rangle
\end{array}
$$

Action *Phase*1$a(c,\ r)$ is executed by the coordinator c of round r as specified in the paper. To ensure Liveness, c can only execute this action if it believes to be the leader and either c has received a message related to a round between $\mathit{crnd}[c]$ and r, or it suspects one of the current collision-fast proposers to have failed.

$$
\begin{array}{l}
\mathit{Phase1a}(c,\ r) \triangleq \\
\quad \land\ \mathit{amLeader}[c] \\
\quad \land\ c = \mathit{CoordOf}(r) \\
\quad \land\ \mathit{crnd}[c] \prec r \\
\quad \land\ \lor\ \exists\, \mathit{msg} \in \mathit{msgs} \setminus [\mathit{type} : \{ \text{“propose”} \},\ \mathit{val} : \mathit{Value}] : \\
\qquad\quad \land\ \mathit{crnd}[c] \prec \mathit{msg.rnd} \\
\qquad\quad \land\ \mathit{msg.rnd} \prec r \\
\qquad \lor\ \land\ \neg(\mathit{CfProposer}(\mathit{crnd}[c]) \subseteq \mathit{activep}[c]) \\
\qquad\quad \land\ \mathit{CfProposer}(r) \subseteq \mathit{activep}[c] \\
\quad \land\ \mathit{crnd}' = [\mathit{crnd}\ \text{EXCEPT}\ ![c] = r] \\
\quad \land\ \mathit{cval}' = [\mathit{cval}\ \text{EXCEPT}\ ![c] = \mathit{none}] \\
\quad \land\ \mathit{Send}([\mathit{type} \mapsto \text{“1a”},\ \mathit{rnd} \mapsto r]) \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{aVars},\ \mathit{pVars},\ \mathit{oVars} \rangle
\end{array}
$$

Action *Phase*1$b(a,\ r)$ is executed by acceptor a, for round r. It follows exactly what is explained in Section 4.1.

$$
\begin{array}{l}
\mathit{Phase1b}(a,\ r) \triangleq \\
\quad \land\ [\mathit{type} \mapsto \text{“1a”},\ \mathit{rnd} \mapsto r] \in \mathit{msgs} \\
\quad \land\ \mathit{rnd}[a] \prec r \\
\quad \land\ \mathit{rnd}' = [\mathit{rnd}\ \text{EXCEPT}\ ![a] = r] \\
\quad \land\ \mathit{Send}([\mathit{type} \mapsto \text{“1b”},\ \mathit{rnd} \mapsto r,\ \mathit{acc} \mapsto a, \\
\qquad\qquad\qquad\qquad \mathit{vrnd} \mapsto \mathit{vrnd}[a],\ \mathit{vval} \mapsto \mathit{vval}[a]]) \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{vrnd},\ \mathit{vval},\ \mathit{pVars},\ \mathit{cVars},\ \mathit{oVars} \rangle
\end{array}
$$

DistProvedSafe$(Q,\ r,\ 1\mathit{bMsg})$ returns a safe initial v-mapping for round r based on the "1b" messages for r sent by acceptors in quorum Q. It returns *Bottom* if no v-mapping has been or might be chosen in a lower-numbered round or a complete v-mapping that extends any v-mapping possibly chosen in a lower-numbered round.

$$
\begin{array}{l}
DistProvedSafe(Q,\, r,\, 1bMsg) \triangleq \\
\text{IF } \forall\, a \in Q : 1bMsg[a].vval = none \\
\quad \text{THEN } Bottom \\
\quad \text{ELSE LET } k \triangleq Max(\{1bMsg[a].vrnd : a \in Q\}) \\
\qquad\qquad AS \triangleq \{a \in Q : \land\, 1bMsg[a].vrnd = k \\
\qquad\qquad\qquad\qquad\qquad \land\, 1bMsg[a].vval \neq none\} \\
\qquad\qquad S \triangleq \{1bMsg[a].vval : a \in AS\} \\
\qquad \text{IN } NilExtension(LUB(S),\, Proposer)
\end{array}
$$

The action $Phase2Start(c,\, r)$ follows the description given in Section 4.1. However, it is only executed if c believes to be the current leader.

$$
\begin{array}{l}
Phase2Start(c,\, r) \triangleq \\
\land\, crnd[c] = r \\
\land\, cval[c] = none \\
\land\, amLeader[c] \\
\land\, \exists\, Q \in Quorum(r) : \\
\quad \land\, \forall\, a \in Q : \exists\, msg \in msgs : \land\, msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\, msg.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\, msg.acc = a \\
\quad \land \text{ LET } 1bMsg \triangleq [a \in Q \mapsto \text{CHOOSE } msg \in msgs : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc = a] \\
\qquad\quad v \triangleq DistProvedSafe(Q,\, r,\, 1bMsg) \\
\qquad \text{IN } \land\, cval' = [cval \text{ EXCEPT } ![c] = v] \\
\qquad\quad\ \land\, Send([type \mapsto \text{“2S”},\, rnd \mapsto r,\, val \mapsto v]) \\
\land \text{ UNCHANGED } \langle aVars,\, pVars,\, crnd,\, oVars \rangle
\end{array}
$$

$Phase2Prepare(p,\, r)$ simply follows the description given in Section 4.1.

$$
\begin{array}{l}
Phase2Prepare(p,\, r) \triangleq \\
\land\, prnd[p] \prec r \\
\land\, \exists\, v \in ValMap : \\
\quad \land\, [type \mapsto \text{“2S”},\, rnd \mapsto r,\, val \mapsto v] \in msgs \\
\quad \land\, \lor\, \land\, v = Bottom \\
\qquad\qquad \land\, pval' = [pval \text{ EXCEPT } ![p] = none] \\
\qquad\ \lor\, \land\, v \neq Bottom \\
\qquad\qquad \land\, pval' = [pval \text{ EXCEPT } ![p] = v[p]] \\
\land\, prnd' = [prnd \text{ EXCEPT } ![p] = r] \\
\land \text{ UNCHANGED } \langle aVars,\, cVars,\, oVars \rangle
\end{array}
$$

Action $Phase2a(p,\, r,\, V)$ also just follows the description of Section 4.1.

```
Phase2a(p, r, V) ≜
 ∧ prnd[p] = r
 ∧ p ∈ CfProposer(r)
 ∧ pval[p] = none
 ∧ ∨ [type ↦ "propose", val ↦ V] ∈ msgs
   ∨ ∧ V = Nil
     ∧ ∃ q ∈ CfProposer(r), U ∈ Value :
         [type ↦ "2a", rnd ↦ r, val ↦ SM(q, U)] ∈ msgs
 ∧ pval' = [pval EXCEPT ![p] = V]
 ∧ Send([type ↦ "2a", rnd ↦ r, val ↦ SM(p, V)])
 ∧ UNCHANGED ⟨prnd, aVars, cVars, oVars⟩
```

The same for action $Phase2b(a, r)$.

```
Phase2b(a, r) ≜
 ∧ rnd[a] ⪯ r
 ∧ ∃ v ∈ ValMap :
   ∧ ∨ ∧ [type ↦ "2S", rnd ↦ r, val ↦ v] ∈ msgs
       ∧ IsValued(v)
       ∧ vrnd[a] ≺ r ∨ vval[a] = none
     ∨ ∃ s ∈ SingleMap :
       ∧ [type ↦ "2a", rnd ↦ r, val ↦ s] ∈ msgs
       ∧ s.v ≠ Nil
       ∧ ∨ ∧ vrnd[a] ≺ r ∨ vval[a] = none
           ∧ v = NilExtension(Bottom • s, Proposer \ CfProposer(r))
         ∨ ∧ vrnd[a] = r ∧ vval[a] ≠ none
           ∧ v = vval[a] • s
   ∧ vval' = [vval EXCEPT ![a] = v]
   ∧ rnd'  = [rnd  EXCEPT ![a] = r]
   ∧ vrnd' = [vrnd EXCEPT ![a] = r]
   ∧ Send([type ↦ "2b", rnd ↦ r, acc ↦ a, val ↦ v])
 ∧ UNCHANGED ⟨pVars, cVars, oVars⟩
```

The *Learn* action is defined differently from the explanation in Section 4.1. Here, for simplicity, we let the value being merged with *learned*[l] be an action parameter.

```
Learn(l, v) ≜
 ∧ ∃ r ∈ RNum :
     ∃ Q ∈ Quorum(r), P ∈ SUBSET CfProposer(r) :
       ∧ ∀ p ∈ P : [type ↦ "2a", rnd ↦ r, val ↦ SM(p, Nil)] ∈ msgs
       ∧ ∀ a ∈ Q : ∃ msg ∈ msgs : ∧ msg.type = "2b"
                                  ∧ msg.rnd  = r
```

$\land\ msg.acc\ = a$
$\land\ v \sqsubseteq NilExtension(msg.val,\ P)$
$\land\ learned' = [learned\ \text{EXCEPT}\ ![l] = @ \sqcup v]$
$\land\ \text{UNCHANGED}\ \langle aVars,\ pVars,\ cVars,\ proposed,\ msgs,$
$noncrashed,\ amLeader,\ activep\rangle$

Message Loss/Retransmission Actions

The following operator returns the last message sent by coordinator c

$CoordLastMsg(c) \triangleq$
IF $cval[c] = none$
THEN $[type \mapsto$ "1a", $rnd \mapsto crnd[c]]$
ELSE $[type \mapsto$ "2S", $rnd \mapsto crnd[c],\ val \mapsto cval[c]]$

The following operator returns the last message sent by proposer p. It is sound only if $pval[p] \neq none$, $crnd[CoordOf(prnd[p])] = prnd[p]$, and $cval[CoordOf(prnd[p])] = Bottom$. This condition is true only if p has fast-proposed a value at round $prnd[p]$.

$ProposerLastMsg(p) \triangleq$
$[type \mapsto$ "2a", $rnd \mapsto prnd[p],\ val \mapsto SM(p,\ pval[p])]$

The following operator returns the last message sent by acceptor a.

$AcceptorLastMsg(a) \triangleq$
IF $vrnd[a] = rnd[a]$
THEN $[type \mapsto$ "2b", $rnd \mapsto rnd[a],\ acc \mapsto a,\ val \mapsto vval[a]]$
ELSE $[type \mapsto$ "1b", $rnd \mapsto rnd[a],\ acc \mapsto a,$
$vrnd \mapsto vrnd[a],\ vval \mapsto vval[a]]$

$LoseMsg(m)$ implements the loss of message m. Any message may be lost except for: - a "propose" message: Indeed the algorithm is resilient to their loss, but implementing retransmission of a "propose" message would make our specification needlessly more complicated.
- the last message sent by a good coordinator that believes to be the leader
- the fast-proposal sent by a good proposer for its current round.
- the last message sent by a good acceptor.
Therefore, the algorithm only requires that such messages be always available for the agents they were sent to.

$LoseMsg(m) \triangleq$
$\land \neg \lor m.type \in \{$"propose"$\}$
$\lor \land m.type \in \{$"1a", "2S"$\}$
$\land m = CoordLastMsg(CoordOf(m.rnd))$
$\land CoordOf(m.rnd) \in noncrashed$

$\land\ amLeader[CoordOf(m.rnd)]$

$\lor\ \land\ m.type \in \{\text{“2a”}\}$
$\land\ pval[m.val.p] \neq none$
$\land\ crnd[CoordOf(m.rnd)] = m.rnd$
$\land\ cval[CoordOf(m.rnd)] = Bottom$
$\land\ m = ProposerLastMsg(m.val.p)$
$\land\ m.val.p \in noncrashed$

$\lor\ \land\ m.type \in \{\text{“1b”}, \text{“2b”}\}$
$\land\ m = AcceptorLastMsg(m.acc)$
$\land\ m.acc \in noncrashed$

$\land\ msgs' = msgs \setminus \{m\}$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

Even though the *LoseMsg*(m) action above does not apply for some messages, retransmission is necessary because an agent can fail, have its message lost, and then recover. In this case it is necessary to re-send the lost message. Below you will find the actions responsible for retransmission.

$CoordRetransmit(c) \triangleq$
$\land\ amLeader[c]$
$\land\ Send(CoordLastMsg(c))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

$AcceptorRetransmit(a) \triangleq$
$\land\ Send(AcceptorLastMsg(a))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

$ProposerRetransmit(p) \triangleq$
$\land\ pval[p] \neq none$
$\land\ crnd[CoordOf(prnd[p])] = prnd[p]$
$\land\ cval[CoordOf(prnd[p])] = Bottom$
$\land\ Send(ProposerLastMsg(p))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

Other Actions

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 specifiction makes no assumption about the outcome of leader selection. (However, progress is guaranteed only under an assumption about the values of *amLeader*[c].)

$LeaderSelection \triangleq$
$\land\ amLeader' \in [Coord \to$ BOOLEAN $]$

$\land$ UNCHANGED $\langle aVars,\ cVars,\ pVars,\ proposed,\ learned,$
$noncrashed,\ activep,\ msgs\rangle$

Action *SuspectOrTrust* arbitrarily changes the values of $activep[c]$, for all coordinators c.

$SuspectOrTrust \triangleq$
$\land\ activep' \in [Coord \to \text{SUBSET}\ Proposer]$
$\land$ UNCHANGED $\langle aVars,\ cVars,\ pVars,\ proposed,\ learned,$
$noncrashed,\ amLeader,\ msgs\rangle$

Action *FailOrRecover* also allows an arbitrary change to the value of *noncrashed*.

$FailOrRecover \triangleq$
$\land\ noncrashed' \in \text{SUBSET}\ (Acceptor \cup Coord \cup Proposer \cup Learner)$
$\land$ UNCHANGED $\langle aVars,\ cVars,\ pVars,\ proposed,\ learned,$
$amLeader,\ activep,\ msgs\rangle$

Final Specification

CoordNext(c) specifies the execution of some action by coordinator c.

$CoordNext(c) \triangleq$
$\lor \exists\, r \in RNum :\ \lor Phase1a(c,\ r)$
$\lor Phase2Start(c,\ r)$
$\lor\ CoordRetransmit(c)$

ProposerNext(p) specifies the execution of some action by proposer p.

$ProposerNext(p) \triangleq$
$\lor \exists\, r \in RNum,\ V \in Value \cup \{Nil\} : Phase2a(p,\ r,\ V)$
$\lor\ ProposerRetransmit(p)$

AcceptorNext(a) specifies the execution of some action by acceptor a.

$AcceptorNext(a) \triangleq$
$\lor \exists\, r \in RNum :\ \lor Phase1b(a,\ r)$
$\lor Phase2b(a,\ r)$
$\lor\ AcceptorRetransmit(a)$

LearnerNext(a) specifies the execution of some action by learner l.

$LearnerNext(l) \triangleq$
$\exists\, v \in ValMap : Learn(l,\ v)$

Next defines the next-state action of the specification.

$Next \triangleq\ \lor \exists\, V \in Value : Propose(V)$

$$
\begin{array}{l}
\lor\, \exists\, c \ \in \mathit{Coord} \quad \cap\, \mathit{noncrashed} : \mathit{CoordNext}(c) \\
\lor\, \exists\, p \ \in \mathit{Proposer} \cap\, \mathit{noncrashed} : \mathit{ProposerNext}(p) \\
\lor\, \exists\, a \ \in \mathit{Acceptor} \cap\, \mathit{noncrashed} : \mathit{AcceptorNext}(a) \\
\lor\, \exists\, l \ \in \mathit{Learner} \ \cap\, \mathit{noncrashed} : \mathit{LearnerNext}(l) \\
\lor\, \exists\, m \in \mathit{msgs} : \mathit{LoseMsg}(m) \\
\lor\, \mathit{LeaderSelection} \lor \mathit{SuspectOrTrust} \lor \mathit{FailOrRecover}
\end{array}
$$

$$\mathit{vars} \triangleq \langle \mathit{aVars},\ \mathit{pVars},\ \mathit{cVars},\ \mathit{oVars},\ \mathit{msgs} \rangle$$

$$
\begin{array}{l}
\mathit{Fairness} \triangleq \\
\quad \land\, \forall\, c \in \mathit{Coord} : \\
\qquad \land\, \mathrm{WF}_{\mathit{vars}}(c \in \mathit{noncrashed} \land \mathit{CoordNext}(c)) \\
\qquad \land\, \mathrm{WF}_{\mathit{vars}}(c \in \mathit{noncrashed} \land (\exists\, r \in \mathit{RNum} : \mathit{Phase1a}(c,\ r))) \\
\quad \land\, \forall\, p \in \mathit{Proposer} : \mathrm{WF}_{\mathit{vars}}(p \in \mathit{noncrashed} \land \mathit{ProposerNext}(p)) \\
\quad \land\, \forall\, a \in \mathit{Acceptor} : \mathrm{WF}_{\mathit{vars}}(a \in \mathit{noncrashed} \land \mathit{AcceptorNext}(a)) \\
\quad \land\, \forall\, l \in \mathit{Learner} : \mathrm{WF}_{\mathit{vars}}(l \in \mathit{noncrashed} \land \mathit{LearnerNext}(l))
\end{array}
$$

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

The theorems below asserts that the spec ensures the type invariant and implements the safety part of the *MConsensus* specification.

THEOREM $\mathit{Spec} \Rightarrow \Box\, \mathit{TypeInv}$

$$\mathit{MC} \triangleq \text{INSTANCE } \mathit{MConsensus}$$

THEOREM $\mathit{Spec} \Rightarrow \mathit{MC}!\mathit{Spec}$

$\mathit{LA}(l,\ c,\ Q)$ defines the liveness assumption required by the algorithm. Since our specification does not allow a "propose" message to be lost, LA is slightly simpler than what we described in the paper.

$$
\begin{array}{l}
\mathit{LA}(l,\ c,\ Q) \triangleq \\
\quad \land\, \{c,\ l\} \cup Q \subseteq \mathit{noncrashed} \\
\quad \land\, \mathit{proposed} \neq \{\} \\
\quad \land\, \forall\, c2 \in \mathit{Coord} : \mathit{amLeader}[c2] \equiv (c = c2) \\
\quad \land\, \mathit{activep}[c] \subseteq \mathit{noncrashed} \\
\quad \land\, \forall\, r \in \mathit{RNum} : \\
\qquad \exists\, r2 \in \mathit{RNum} : \ \land\, r \prec r2 \\
\qquad\qquad\qquad\qquad\ \land\, c = \mathit{CoordOf}(r2) \\
\qquad\qquad\qquad\qquad\ \land\, \mathit{CfProposer}(r2) \subseteq \mathit{activep}[l] \\
\quad \land\, \mathit{activep}[c] \subseteq \mathit{activep}[c]'
\end{array}
$$

The theorem below asserts that the algorithm's specification satisfies Liveness if the liveness assumption eventually holds forever.

THEOREM $\forall\, l \in \mathit{Learner} :$
$\quad \wedge\, \mathit{Spec}$
$\quad \wedge\, \exists\, Q \in \text{SUBSET } \mathit{Acceptor} :$
$\quad\quad \wedge\, \forall\, r \in \mathit{RNum} : Q \in \mathit{Quorum}(r)$
$\quad\quad \wedge\, \exists\, c \in \mathit{Coord} : \Diamond\Box[\mathit{LA}(l,\, c,\, Q)]_{\mathit{vars}}$
$\quad \Rightarrow \Diamond(\text{DOMAIN } \mathit{learned}[l] = \mathit{Proposer})$

D.9 Collision-Fast Atomic Broadcast

MODULE *CFPaxosAbcast*

This module presents the specification of an efficient collision-fast atomic broadcast algorithm based on the Collision-fast *Paxos* algorithm for M-Consensus.

EXTENDS *Naturals*, *Sequences* Definitions concerning the natural numbers and sequences.

RNum is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called *Zero*.

CONSTANTS RNum, $\_ \preceq \_$, Zero

The definition of $i \prec j$

$i \prec j \triangleq (i \preceq j) \wedge (i \neq j)$

The specification has the same parameters as the Collision-Fast *Paxos* algorithm.

CONSTANTS $\mathit{Proposer}$, $\mathit{Learner}$, Coord, $\mathit{CoordOf}(\_)$, $\mathit{Acceptor}$,
$\mathit{Quorum}(\_)$, $\mathit{CfProposer}(\_)$, Value, Nil, none

The Atomic Broadcast algorithm is based on an infinite number of Collision-fast *Paxos* (*CFPaxos*) instances. However, some variables such as proposed, *rnd*, *prnd*, *crnd*, *noncrashed*, *amLeader*, and *activep* are shared by all instances. The other variables are implemented by arrays indexed by the instance. We represent this arrays by the original variable name with the prefix x as it can be seen below.

VARIABLES $\mathit{proposed}$, $\mathit{xlearned}$, rnd, xvrnd, xvval, prnd, xpval,
crnd, xcval, xmsgs, $\mathit{noncrashed}$, $\mathit{amLeader}$, $\mathit{activep}$

vars is a sequence containing all the specification variables.

$\mathit{vars} \triangleq \langle \mathit{proposed}, \mathit{xlearned}, \mathit{rnd}, \mathit{xvrnd}, \mathit{xvval}, \mathit{prnd}, \mathit{xpval}, \mathit{crnd}, \mathit{xcval},$
$\mathit{xmsgs}, \mathit{noncrashed}, \mathit{amLeader}, \mathit{activep} \rangle$

CFP(i), where i is a Natural number, is the Collision-fast *Paxos* instance number i.

$\mathit{CFP}(i) \triangleq \text{INSTANCE } \mathit{DistCFPaxosLiv} \text{ WITH } \mathit{learned} \leftarrow \mathit{xlearned}[i],$
$\mathit{vrnd} \leftarrow \mathit{xvrnd}[i],$

$$
\begin{array}{ll}
vval & \leftarrow xvval[i], \\
pval & \leftarrow xpval[i], \\
msgs & \leftarrow xmsgs[i], \\
cval & \leftarrow xcval[i]
\end{array}
$$

Initial state of the specification

$$
\begin{array}{ll}
Init \triangleq & \land\, proposed = \{\} \\
& \land\, xlearned = [i \in Nat \mapsto [l \in Learner \mapsto CFP(i)!Bottom]] \\
& \land\, rnd = [a \in Acceptor \mapsto Zero] \\
& \land\, xvrnd = [i \in Nat \mapsto [a \in Acceptor \mapsto Zero]] \\
& \land\, xvval = [i \in Nat \mapsto [a \in Acceptor \mapsto none]] \\
& \land\, prnd = [p \in Proposer \mapsto Zero] \\
& \land\, xpval = [i \in Nat \mapsto [p \in Proposer \mapsto none]] \\
& \land\, crnd = [c \in Coord \mapsto Zero] \\
& \land\, xcval = [i \in Nat \mapsto [c \in Coord \mapsto \text{IF } c = CoordOf(Zero) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } CFP(i)!Bottom \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none]] \\
& \land\, xmsgs = [i \in Nat \mapsto \{\}] \\
& \land\, noncrashed = Acceptor \cup Coord \cup Proposer \cup Learner \\
& \land\, amLeader = [c \in Coord \mapsto \text{IF } c = CoordOf(Zero) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{THEN TRUE} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE FALSE}] \\
& \land\, activep = [c \in Coord \mapsto Proposer]
\end{array}
$$

Some actions of the algorithm have to do only with one instance of *CFPaxos*. For such cases it is interesting to have an operator that keeps non-shared variables of other instances unchanged.

$$
\begin{array}{l}
InstanceUnchanged(i) \triangleq \\
\quad \text{UNCHANGED } \langle xlearned[i],\ xvrnd[i],\ xvval[i], \\
\qquad\qquad xpval[i],\ xcval[i],\ xmsgs[i]\rangle
\end{array}
$$

Action $Phase1a(c,\ r)$ for instance i must be slightly changed so that a new round might be started if there is an interfering message for a different round number in ANY of the running instances.

$$
\begin{array}{l}
NewPhase1a(i,\ c,\ r) \triangleq \\
\quad \land\, amLeader[c] \\
\quad \land\, c = CoordOf(r) \\
\quad \land\, crnd[c] \prec r \\
\quad \land\, \lor\, \exists\, j \in Nat : \\
\qquad\quad \exists\, msg \in xmsgs[j] : \\
\qquad\qquad \land\, msg.type \neq \text{“propose”}
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ crnd[c] \prec msg.rnd \\
\quad \wedge\ msg.rnd \prec r \\
\vee\ \wedge\ \neg(CfProposer(crnd[c]) \subseteq activep[c]) \\
\quad \wedge\ CfProposer(r) \subseteq activep[c] \\
\wedge\ crnd' = [crnd \text{ EXCEPT } ![c] = r] \\
\wedge\ xcval'[i] = [xcval[i] \text{ EXCEPT } ![c] = none] \\
\wedge\ CFP(i)!Send([type \mapsto \text{“1a”},\ rnd \mapsto r]) \\
\wedge\ \text{UNCHANGED } \langle CFP(i)!aVars,\ CFP(i)!pVars,\ CFP(i)!oVars\rangle
\end{array}
$$

Agent Actions

We now present the combinated actions each agent performs.

A *Propose* action sends a “propose” message that is valid for all the *CFPaxos* instances. This is logically implemented by executing a propose action for each instance. Notice, however, that the multiple logical “propose” messages are implemented by a single one in practice.

$$
\begin{array}{l}
Propose(V) \triangleq \\
\quad \forall\, i \in Nat : CFP(i)!Propose(V)
\end{array}
$$

Action *Phase*1*a* executes *NewPhase*1*a* specified above for all *CFPaxos* instances. It is easy to see that, since all pre-conditions of the action are based on variables shared among all instances, the action is enable for instance i iff it is enabled for instance j. As a result of this action, a “1a” message is sent for each instance. In practice, a single “1a” message is sent and it is interpreted as valid for all instances.

$$
\begin{array}{l}
Phase1a(c,\ r) \triangleq \\
\quad \forall\, i \in Nat : NewPhase1a(i,\ c,\ r)
\end{array}
$$

Action $Phase1b(a,\ r)$ is executed when acceptor a receives the “1a” message for a higher-numbered round than its current one. Since there is a logical message for each instance, the *Phase*1*b* action of every instance is equally enabled and are executed. However, different instances might generate different “1b” messages. These different logical messages are sent on the same physical one. The size of this message can be limited because only a finite number of (initial) instances will result on “1b” messages different from $\langle$“1b”, r, a, *Zero*, *none*$\rangle$.

$$
\begin{array}{l}
Phase1b(a,\ r) \triangleq \\
\quad \forall\, i \in Nat : CFP(i)!Phase1b(a,\ r)
\end{array}
$$

Action $Phase2Start(c,\ r)$ is executed by the coordinator of r when it receives the composite "$1b$" message of the previous action. The coordinator calculates the different "$2S$" messages for every instance. These "$2S$" messages are sent together in the same physical one as we have done in the previous action. Recall that only a finite number of instances will result on "$2S$" messages different from $\langle$"$2S$", r, $Bottom\rangle$, which can be used to limit the size of the composite "$2S$" message sent.

$Phase2Start(c,\ r) \triangleq$
$\forall\, i \in Nat : CFP(i)!Phase2Start(c,\ r)$

Action $Phase2Prepare(p,\ r)$ is executed by proposer p when it receives the composite "$2S$" message from the action above. The action executes $Phase2Prepare$ for every *CFPaxos* instance.

$Phase2Prepare(p,\ r) \triangleq$
$\forall\, i \in Nat : CFP(i)!Phase2Prepare(p,\ r)$

Action $Phase2a(p,\ r,\ V)$ is executed by proposer p when it fast-proposes value V. It executes $Phase2a(p,\ r,\ V)$ for some *CFPaxos* instance i, as long as p has not fast-proposed the same value for a different instance. All the other instances are left unchanged.

$Phase2a(p,\ r,\ V) \triangleq$
$\exists\, i \in Nat :$
$\quad \land\ V \notin \{xpval[j][p] : j \in Nat\}$
$\quad \land\ CFP(i)!Phase2a(p,\ r,\ V)$
$\quad \land\ \forall\, j \in Nat : j < i \Rightarrow xpval[j][p] \neq none$
$\quad \land\ \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Action $Phase2b(a,\ r)$ executes $Phase2b(a,\ r)$ for some *CFPaxos* instance i.

$Phase2b(a,\ r) \triangleq$
$\exists\, i \in Nat :$
$\quad \land\ CFP(i)!Phase2b(a,\ r)$
$\quad \land\ \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Action $Learn(l,\ v)$ executes $Learn(l,\ v)$ for some *CFPaxos* instance i.

$Learn(l,\ v) \triangleq$
$\exists\, i \in Nat :$
$\quad \land\ CFP(i)!Learn(l,\ v)$
$\quad \land\ \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Message Loss/Retransmission Actions

LoseMsg(m) takes into consideration the fact that messages "$1a$", "$2S$", and "$1b$" are composite, with a logical message for every instance but with all grouped in the same physical message. As a result, all grouped messages must be lost together. The other sorts of message are not composite and can be lost in a single instance only.

$$
\begin{array}{l}
LoseMsg(m) \triangleq \\
\lor \land m.type \in \{\text{“1a”}, \text{“2S”}\} \\
\quad \land \forall\, i \in Nat : \\
\quad\quad \land \exists\, m2 \in CFP(i)!Msg : \\
\quad\quad\quad \land m2.type = m.type \\
\quad\quad\quad \land m2.rnd \;= m.rnd \\
\quad\quad\quad \land CFP(i)!LoseMsg(m2) \\
\\
\lor \land m.type = \text{“1b”} \\
\quad \land \forall\, i \in Nat : \\
\quad\quad \land \exists\, m2 \in CFP(i)!Msg : \\
\quad\quad\quad \land m2.type = m.type \\
\quad\quad\quad \land m2.rnd \;= m.rnd \\
\quad\quad\quad \land m2.acc \;= m.acc \\
\quad\quad\quad \land CFP(i)!LoseMsg(m2) \\
\\
\lor \land m.type \in \{\text{“2a”}, \text{“2b”}\} \\
\quad \land \exists\, i \in Nat : \\
\quad\quad \land CFP(i)!LoseMsg(m) \\
\quad\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)
\end{array}
$$

By the way coordinator actions are grouped for all the instances, its last message is allways of the same type and for the same round. Therefore, retransmission boils down to just retransmitting the last message in every instance.

$$
\begin{array}{l}
CoordRetransmit(c) \triangleq \\
\forall\, i \in Nat : CFP(i)!CoordRetransmit(c)
\end{array}
$$

If the last logical message of an acceptor for every instance is of type "$1b$", it means that its last action has been a $Phase1b(a,\, r)$ for some round r. In this case, it is not clear whether the coordinator of r has received the composite "$1b$" message from a or not, so a resends it. If it is not the case that the last logical message of a for every instance is of type "$1b$", then a has accepted some value for its current round and this means that the coordinator does not need its composite "$1b$" message. So, a only re-sends the "$2b$" messages for the instances at which it has accepted some value and leave the other instances unchanged.

$$
\begin{array}{l}
AcceptorRetransmit(a) \triangleq \\
\lor \forall\, i \in Nat : \\
\quad\quad \land CFP(i)!AcceptorLastMsg(a).type = \text{“1b”} \\
\quad\quad \land CFP(i)!AcceptorRetransmit(a)
\end{array}
$$

$\lor \land \exists\, i \in Nat : CFP(i)!AcceptorLastMsg(a).type = \text{“2b”}$
$\quad \land \forall\, i \in Nat :$
$\qquad$ IF $CFP(i)!AcceptorLastMsg(a).type = \text{“2b”}$
$\qquad\quad$ THEN $CFP(i)!AcceptorRetransmit(a)$
$\qquad\quad$ ELSE $InstanceUnchanged(i)$

A proposer retransmission is only a single retransmission for some *CFPaxos* instance i.

$ProposerRetransmit(p) \triangleq$
$\exists\, i \in Nat :$
$\quad \land CFP(i)!ProposerRetransmit(p)$
$\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

Other Actions

LeaderSelection, *SuspectOrTrust*, and *FailOrRecover* just execute the actions with the same name on some instance i and leave the other instances unchanged. These actions actually influence all the instances because they deal with shared variables.

$LeaderSelection \triangleq$
$\exists\, i \in Nat :$
$\quad \land CFP(i)!LeaderSelection$
$\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

$SuspectOrTrust \triangleq$
$\exists\, i \in Nat :$
$\quad \land CFP(i)!SuspectOrTrust$
$\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

$FailOrRecover \triangleq$
$\exists\, i \in Nat :$
$\quad \land CFP(i)!FailOrRecover$
$\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

Final Specification

CoordNext(c) specifies the execution of some action by coordinator c.

$CoordNext(c) \triangleq$
$\lor \exists\, r \in RNum : \lor Phase1a(c,\, r)$
$\qquad\qquad\qquad\quad \lor Phase2Start(c,\, r)$
$\lor CoordRetransmit(c)$

ProposerNext(p) specifies the execution of some action by proposer p.

$$
\begin{array}{l}
ProposerNext(p) \triangleq \\
\quad \lor \exists\, r \in RNum,\ V \in Value \cup \{Nil\} : Phase2a(p,\ r,\ V) \\
\quad \lor ProposerRetransmit(p)
\end{array}
$$

AcceptorNext(a) specifies the execution of some action by acceptor a.

$$
\begin{array}{l}
AcceptorNext(a) \triangleq \\
\quad \lor \exists\, r \in RNum : \lor Phase1b(a,\ r) \\
\qquad\qquad\qquad\ \ \lor Phase2b(a,\ r) \\
\quad \lor AcceptorRetransmit(a)
\end{array}
$$

LearnerNext(a) specifies the execution of some action by learner l.

$$
\begin{array}{l}
LearnerNext(l) \triangleq \\
\quad \exists\, v \in CFP(0)!ValMap : Learn(l,\ v)
\end{array}
$$

Next defines the next-state action of the specification.

$$
\begin{array}{lll}
Next \triangleq & \lor \exists\, V \in Value & : Propose(V) \\
& \lor \exists\, c \in Coord \cap noncrashed & : CoordNext(c) \\
& \lor \exists\, p \in Proposer \cap noncrashed & : ProposerNext(p) \\
& \lor \exists\, a \in Acceptor \cap noncrashed & : AcceptorNext(a) \\
& \lor \exists\, l \in Learner \cap noncrashed & : LearnerNext(l) \\
& \lor \exists\, m \in \text{UNION}\ \{xmsgs[i] : i \in Nat\} & : LoseMsg(m) \\
& \lor LeaderSelection \lor SuspectOrTrust \lor FailOrRecover &
\end{array}
$$

The fairness condition of the specification. We need weak fairness on the agent actions for every instance, since this is part of the liveness requirement for a single instance of *CFPaxos*.

$$
\begin{array}{l}
Fairness \triangleq \\
\quad \land \forall\, c \in Coord : \\
\qquad \land \mathrm{WF}_{vars}(c \in noncrashed \land CoordNext(c)) \\
\qquad \land \mathrm{WF}_{vars}(c \in noncrashed \land (\exists\, r \in RNum : Phase1a(c,\ r))) \\
\quad \land \forall\, p \in Proposer,\ i \in Nat : \\
\qquad \mathrm{WF}_{vars}(p \in noncrashed \land CFP(i)!ProposerNext(p)) \\
\quad \land \forall\, a \in Acceptor,\ i \in Nat : \\
\qquad \mathrm{WF}_{vars}(a \in noncrashed \land CFP(i)!AcceptorNext(a)) \\
\quad \land \forall\, l \in Learner,\ i \in Nat : \\
\qquad \mathrm{WF}_{vars}(l \in noncrashed \land CFP(i)!LearnerNext(l))
\end{array}
$$

The final specification

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

Below we define the interface mapping from the algorithm above and the specification of Atomic Broadcast.

The set of broadcast messages is simply our set of proposed values.

$broadcast \triangleq proposed$

We assume the set *Proposer* can be totally ordered by a mapping *POrd* that matches each proposer to a natural number and no two proposers to the same one.

ASSUME $\exists\, POrd \in [Proposer \to Nat] :$
$\qquad \forall\, p,\, q \in Proposer : p \neq q \Rightarrow POrd[p] \neq POrd[q]$

$POrd \triangleq$ CHOOSE $POrd \in [Proposer \to Nat] :$
$\qquad \forall\, p,\, q \in Proposer : p \neq q \Rightarrow POrd[p] \neq POrd[q]$

The array delivered that maps each learner to its learned sequence is defined below, in terms of the instances that have already been terminated and the last partially terminated sequence.

$delivered \triangleq$
LET $defined[m \in ValMap,\, s \in Seq(Values)] \triangleq$
LET $defSet \triangleq \{p \in$ DOMAIN $m :$
$\qquad \wedge\, m[p] \neq Nil$
$\qquad \wedge \neg\exists\, i \in$ DOMAIN $s : s[i] = m[p]$
$\qquad \wedge\, \forall\, q \in$ DOMAIN $m : POrd[q] < POrd[p] \Rightarrow m[q] \neq m[p]$
$\qquad \wedge\, \forall\, q \in Proposer \quad : POrd[q] < POrd[p] \Rightarrow q \in$ DOMAIN $m\}$

IN CHOOSE $f \in [1\,..\,Cardinality(defSet) \to defSet] :$
$\qquad \forall\, i,\, j \in$ DOMAIN $f : i \leq j \equiv POrd[f[i]] \leq POrd[f[j]]$

$deliver[l \in Learner,\, i \in Nat,\, s \in Seq(Value)] \triangleq$
IF $Domain =$ DOMAIN $xlearned[i][l]$
THEN $deliver[l,\, i+1,\, s \circ defined[xlearned[i][l],\, s]]$
ELSE $s \circ defined[xlearned[i][l],\, s]$

IN $[l \in Learner \mapsto deliver[l,\, 0,\, \langle\rangle]]$

The following theorem asserts that the algorithm's specification implements atomic broadcast.

$AB \triangleq$ INSTANCE $ABcast$
THEOREM $Spec \Rightarrow AB!Spec$

$LA(V,\ l,\ c,\ Q)$ defines the liveness assumption required by the algorithm. It is the same as Collision-fast *Paxos*.

$$
\begin{array}{l}
LA(V,\ l,\ c,\ Q) \triangleq \\
\quad \land \{c,\ l\} \cup Q \subseteq noncrashed \\
\quad \land V \in proposed \\
\quad \land \forall\, c2 \in Coord : amLeader[c2] \equiv (c = c2) \\
\quad \land activep[c] \subseteq noncrashed \\
\quad \land \forall\, r \in RNum : \\
\qquad \exists\, r2 \in RNum : \land r \prec r2 \\
\qquad\qquad\qquad\qquad \land c = CoordOf(r2) \\
\qquad\qquad\qquad\qquad \land CfProposer(r2) \subseteq activep[l] \\
\quad \land activep[c] \subseteq activep[c]'
\end{array}
$$

The theorem below asserts that the algorithm's specification satisfies Liveness if the liveness assumption eventually holds forever.

$$
\begin{array}{l}
\text{THEOREM } \forall\, l \in Learner,\ V \in Value : \\
\qquad \land Spec \\
\qquad \land \exists\, Q \in \text{SUBSET } Acceptor : \\
\qquad\quad \land \forall\, r \in RNum : Q \in Quorum(r) \\
\qquad\quad \land \exists\, c \in Coord\ : \Diamond\Box[LA(V,\ l,\ c,\ Q)]_{vars} \\
\qquad \Rightarrow \Diamond(\exists\, j \in 1\,..\,Len(delivered[l]) : delivered[l][j] = V)
\end{array}
$$