Collision-fast Atomic Broadcast

Rodrigo Schmidt
Facebook, USA
Email: rodrigo@fb.com

Lasaro Camargos
Federal University of Uberlândia, Brazil
Email: lasaro@facom.ufu.br

Fernando Pedone
University of Lugano, Switzerland
Email: fernando.pedone@usi.ch

***Abstract*—Atomic Broadcast, an important abstraction in dependable distributed computing, is usually implemented by solving infinitely 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, those with timely 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.**

I. Introduction

Atomic broadcast is a fundamental abstraction, at the core of state-machine replication [1], [2]. In an atomic broadcast-based implementation of state-machine replication, clients broadcast commands to the replicas, which deliver and deterministically execute the agreed sequence of commands in the same total order. Naturally, the performance of the replicated system hinges on the performance of atomic broadcast. This paper targets a class of atomic broadcast algorithms that can deliver messages within two communication steps. It advances the state-of-the-art by presenting an algorithm that achieves this bound under weaker conditions than existing algorithms.

Atomic broadcast is often solved by means of a totally ordered succession of consensus instances (e.g., [3], [4]). In each consensus instance, one or more *proposers* can propose a value. Consensus ensures that the decision, a value among the proposed ones, is learned by all nonfaulty *learners*. To broadcast a message m, a process p proposes m in the first consensus instance in which p has not proposed or learned any value yet—to know whether their proposal was the value decided or not, proposers must also be learners. The delivery order of atomic broadcast is given by the order of the consensus instances: the i^{th} instance's decision gives the i^{th} element in the delivered sequence. If its proposed message is not decided, the process must propose it again in a different instance.

In consensus algorithms that rely on a *coordinator*, proposals are first sent to the coordinator, which chooses one proposal and tries to get it decided in the first instance it has not used yet. In such algorithms, a message can be learned in three message steps in the general case and in two message steps if proposed by the coordinator. There are consensus algorithms that bypass the coordinator in order to achieve the latency of two message steps for multiple proposers. Getting a proposal decided in two message steps with multiple proposers, however, requires larger quorums (e.g., Fast Paxos [5]). Moreover, the absence of a coordinator may result in *collisions*, which happen when two concurrent proposals are issued but none is decided after two message steps; solving a collision requires extra message steps.

In [6], Lamport presented two conditions under which asynchronous consensus protocols may decide in two-communication steps even in the presence of collisions, along with *collision-fast* protocols that explore these conditions. The first condition restricts fault tolerance to a single failure and is solved by a simple variant of Paxos. The second condition forces all but one of the collision-fast proposers, those whose proposals may be decided in two steps, to also play the role of an *acceptor* (i.e., a process involved in the consensus decision). The presented protocols still have the problem that only one value, among the proposed ones, is decided per consensus instance; proposed values not in the decision must be proposed again in a different instance. In order for multiple broadcast messages to be delivered within two message steps, a consensus decision must happen in a single communication step. The conditions under which such a *hyper-fast learning* can happen [6] are restrictive though: a single failure is tolerated, every learner is an acceptor or the learner plus the single hyper-fast-proposer form a quorum.

To circumvent the limitations discussed above, we reduce atomic broadcast to M-Consensus, a variation of consensus where processes decide on a bounded composite of proposed values, not on a single value. Collision-fast Paxos, our solution to M-Consensus, tolerates as many failures as original Paxos but allows multiple proposers, not only the coordinator, to have their proposal learned in two message steps. Collision-fast Paxos can be used to implement atomic broadcast with a succession of M-Consensus instances, as done in the standard reduction to consensus. Since multiple messages may be part of each Collision-Fast Paxos' decision, multiple messages can be delivered after each instance, in two steps.

The rest of the paper is organized as follows. In Section II, we present the system model. In Sections III and IV, we describe M-Consensus and Collision-Fast Paxos, respectively. In Section V, we show how to implement atomic broadcast with Collision-Fast Paxos. We discuss related work in Section VI and conclude in Section VII. TLA$^+$ specifications and correctness proofs for all of our algorithms are available in the companion technical report [7].

The authors would like to thank PROPP, grant 4/2012-053 for financially supporting this work.

II. Model and Definitions

We assume an asynchronous crash-recovery model in which *agents* (i.e., processes, or threads in a process) 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, and if repeatedly sent by a nonfaulty agent to a nonfaulty agent, then they are eventually delivered. Agents can fail by stopping only and never perform incorrect actions; an agent is considered to be nonfaulty iff it never fails. Agents are assumed to have a local stable storage to keep their state in between failures so that finite periods of absence are not distinguishable from excessive slowness. Although agents may recover, they are not obliged to do so once they have failed.

A. Atomic Broadcast

Given two sets of agents, namely *proposers* and *learners*, atomic broadcast consists in ensuring that messages broadcast by proposers are eventually delivered by 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. We represent a *sequence* s of length n as $\langle v_1, v_2, \ldots, v_n \rangle$, where v_i is 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 all sequences.

Atomic broadcast's safety properties can 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 $delivered[l2] \sqsubseteq delivered[l1]$.

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. Liveness 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.

B. Algorithms

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. Events are totally ordered at the agents performing them, that is, each event e performed by agent e_{agent} is uniquely identified by the positive integer e_{num}, indicating that e was the ${e_{num}}^{th}$ 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. More formally, for any set S of events, let $\preceq_S$ be 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 $f_{rcvd} = \langle e_{msg}, e_{agent}, e_{num} \rangle$. 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.

Definition 1 (Scenario [6]): 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.

Definition 2 (Prefix [6]): 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 is the set of non-empty scenarios it allows. An *asynchronous algorithm* is defined as follows, where $Agents(S)$ is the set of agents that performed events in S.

Definition 3 (Asynchronous Algorithm [6]): 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 [6]): 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.

We now define what a collision-fast atomic broadcast protocol is. For simplicity, our definition considers only normal 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, our algorithm ensures the same delivery latency for all messages broadcast in normal runs.

Definition 5 (Normal Scenario [6]): 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 (i.e., every non-source event is triggered by the receipt of a message);
- 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 [6]): 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.

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 $m_1, \ldots, m_k$.

III. 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, we define the value mapping data structure, v-mapping for short, it depends upon.

A. Value Mapping Sets

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 $Value$. A v-mapping is a function that maps some elements of $Domain$ to either a value in $Value$ or Nil, where Nil is a special value not in $Value$. Formally, $ValMap = \bigcup\{[D \to R] : D \subseteq Domain \wedge R = Value \cup \{Nil\}\}$. Since $\{\} \subseteq Domain$ for any set $Domain$, $\perp$ is in every v-mapping set. 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.

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 $Domain$. Hence, a complete v-mapping does not have any strict extension, since no append operation applied to it can result in a different v-mapping. The complete v-mapping that maps every element in $Domain$ to Nil and, therefore, is independent of the set $Value$ is called the *trivial* v-mapping. A v-mapping is *nontrivial* iff it is different from the trivial one.

B. Problem definition

M-Consensus 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. A v-mapping is *proposed* iff all elements of its domain are mapped either to Nil or to a proposed value. The properties of M-Consensus are defined as follows, where $learned[l]$ represents the v-mapping currently learned by learner l, initially $\perp$.

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.

With respect to solvability, M-Consensus is equivalent to consensus. An algorithm that solves consensus can solve M-Consensus by having learners learn a mapping in which one proposer is mapped to the decided value and all the others are mapped to Nil. An algorithm that solves M-Consensus trivially solves consensus by totally ordering the set of proposers and picking up the value mapped to the first proposer not mapped to Nil as the decision. The advantage of M-Consensus over classical consensus is that it allows two (or more) concurrent proposals to appear in the decision, mapped to different proposers, avoiding the collision of proposals.

IV. Collision-Fast Paxos

A. Basic Algorithm

We first present an algorithm that does not ensure liveness; we address liveness in the next section. The algorithm is structured in rounds, ordered by relation $\leq$. Unless stated otherwise (Section IV-B), rounds correspond to the natural numbers. We assign to every round r a single *coordinator*—a different sort of agent, besides proposers, acceptors, and learners—and a subset of the proposers we call the *collision-fast proposers*. The collision-fast proposers of r are the only proposers that can have their proposals learned in two communication steps in r. As we explain later, making all proposers collision-fast for all rounds would restrict the algorithm's resilience.

At round r, a collision-fast proposer p *fast-proposes* an s-mapping $\langle p, V\rangle$ at most once. It does so 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, a v-mapping is chosen at round r if it is a prefix of every v-mapping accepted by some quorum Q of acceptors 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 received, 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 compatible with v-mappings chosen in other rounds. The algorithm keeps the invariant that if a v-mapping is or might yet 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 a 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 a round).

For the coordinator to be able to identify if some value has been or might be chosen at a lower-numbered round by querying a quorum of acceptors, we need the following assumption. A simple way to ensure Assumption 1 is to define quorums as any majority of the acceptors.

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 [6].

Algorithm 1 presents Collision-fast Paxos in detail. For brevity, we define $a \bullet S$, where S is a set, as the cumulative application of $\bullet$ to the elements of S (in any order). For example, $a \bullet \{x, y, z\} = a \bullet x \bullet y \bullet z$

The algorithm is presented as a set of actions, executed only if all the pre-conditions are satisfied. On normal runs, actions follow the flow depicted in Figure 1.

B. Ensuring Liveness

We now extend Algorithm 1 to ensure progress in case messages are lost, coordinators or collision-fast proposers crash, and coordinators keep on starting new rounds. We assume that if agents a and b do not crash and a keeps sending message m to b, then b eventually receives m. Moreover, we assume weak fairness on the actions an agent may take: no

Algorithm 1 Collision-fast Paxos

Pr: proposers set;
A: acceptors set;
L: learners set;
$CF(i)$: round i's collision-fast proposers set;
$C(i)$: round i's coordinator.
$prnd[p]$, $crnd[c]$, $rnd[a]$: current round of proposer p, coordinator c, and acceptor a, respectively, initially 0.
$pval[p]$: value p has fast-proposed at $prnd[p]$ or $none$ if p has not fast-proposed at $prnd[p]$, initially $none$.
$cval[c]$: initial v-mapping for $crnd[c]$, if c has queried an acceptor quorum or $none$ otherwise; initially $\bot$ for coordinator of round 0 and $none$ for others.
$vrnd[a]$: round at which a has accepted its latest value.
$vval[a]$: v-mapping a has accepted at $vrnd[a]$ or $none$ if no value accepted at $vrnd[a]$; initially $none$.
$learned[l]$: v-mapping currently learned by learner l; initially $\bot$.

1: $Propose(p, V) \triangleq$
2: **pre-condition:**
3: $p \in Pr$
4: **action:**
5: send $\langle$"propose", $V\rangle$ to $cf \in CF(prnd[p])$
6: $Phase1a(c, r) \triangleq$
7: **pre-conditions:**
8: $c = C(r)$
9: $crnd[c] < r$
10: **actions:**
11: $crnd[c] \leftarrow r$
12: $cval[c] \leftarrow none$
13: send $\langle$"1a", $r\rangle$ to A
14: $Phase1b(a, r) \triangleq$
15: **pre-conditions:**
16: $a \in A$
17: $rnd[a] < r$
18: received $\langle$"1a", $r\rangle$ from $C(r)$
19: **actions:**
20: $rnd[a] \leftarrow r$
21: send $\langle$"1b", $a, r, vrnd[a], vval[a]\rangle$ to $C(r)$
22: $Phase2Start(c, r) \triangleq$
23: **pre-conditions:**
24: $c = C(r)$
25: $crnd[c] = r$
26: $cval[c] = none$
27: $\exists Q : Q$ is a quorum: $\forall a \in Q$ received $\langle$"1b", $a, r, vrnd, vval\rangle$
28: **actions:**
29: LET $msgs = [m = \langle$"1b", $a, r, vrnd, vval\rangle$: received m from $a \in Q$]
30: LET $k = Max([vend : \langle$"1b", $a, r, vrnd, vval\rangle \in msgs$])
31: LET $S = [vval : \langle$"1b", $a, r, k, vval\rangle \in m$, $vval \neq none]$
32: IF $S = \emptyset$ THEN
33: $cval[c] \leftarrow \bot$
34: send $\langle$"2S", $r, cval[c]\rangle$ to P
35: ELSE
36: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in Pr]$
37: send $\langle$"2S", $r, cval[c]\rangle$ to $P \cup A$
38: $Phase2Prepare(p, r) \triangleq$
39: **pre-conditions:**
40: $p \in Pr$
41: $prnd[p] < r$
42: received $\langle$"2S", $c, r, v\rangle$
43: **actions:**
44: $prnd[p] \leftarrow r$
45: IF $v = \bot$ THEN $pval[p] \leftarrow none$ ELSE $pval[p] \leftarrow v(p)$
46: $Phase2a(p, r, V) \triangleq$
47: **pre-conditions:**
48: $p \in CF(r)$
49: $prnd[p] = r$
50: $pval[p] = none$
51: **either** $V \neq Nil$ **and** received $\langle$"propose", $p, V\rangle$
52: **or** $V = Nil$ **and** received $\langle$"2a", $r, \langle q, W\rangle\rangle$, $q \in CF(r)$, $W \neq Nil$
53: **actions:**
54: $pval[p] \leftarrow V$
55: **if** $V \neq Nil$ **then** send $\langle$"2a", $p, r, \langle p, V\rangle\rangle$ to $A \cup CF(r)$
56: **else** send $\langle$"2a", $p, r, \langle p, V\rangle\rangle$ to L
57: $Phase2b(a, r) \triangleq$
58: LET $Cond1 =$ (received $\langle$"2S", $r, v\rangle$, $v \neq \bot$ **and** $vrnd[a] < r$) **or** $vval[a] = none$
59: LET $Cond2 =$ received $\langle$"2a", $r, \langle p, V\rangle\rangle$, $V \neq Nil$
60: **pre-conditions:**
61: $a \in A$
62: $rnd[a] \leq r$
63: **either** $Cond1$ **or** $Cond2$
64: **actions:**
65: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
66: IF $Cond1$ THEN $vval[a] \leftarrow v$
67: ELSE IF $Cond2$ **and** ($vrnd[a] < r$ **or** $vval[a] = none$)
68: THEN $vval[a] \leftarrow \bot \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
69: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
70: send $\langle$"2b", $a, r, vval[a]\rangle$ to L
71: $Learn(l) \triangleq$
72: **pre-conditions:**
73: $l \in learners$
74: $\exists Q, Q$ is a quorum: $\forall a \in Q$ received $\langle$"2b", $a, r, \_\rangle$
75: **actions:**
76: LET $P \subset CF(r) : \forall p \in P$ received $\langle$"2a", $p, r, \langle p, Nil\rangle\rangle$
77: $Q2bVals = [v :$ received $\langle$"2b", $a, r, v\rangle$ from $a \in Q$]
78: $w = \sqcap Q2bVals \bullet [\langle p, Nil\rangle : p \in P]$
79: $learned[l] = learned[l] \sqcup w$

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 [9] and the equivalence between consensus and M-Consensus imply that these assumptions are not enough to ensure liveness for M-Consensus. We circumvent FLP by eventually electing a distinguished coordinator—the leader—responsible for starting new rounds. We require that every coordinator be reponsible for infinitely many higher-numbered rounds, which can be done by having round numbers defined as tuples $\langle n, c\rangle$ where n is a natural number and c is its coordinator identifier; thus, round numbers can be compared lexicographically.

When the leader starts a round and picks $\bot$ 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 the Collision-fast Learning theorem

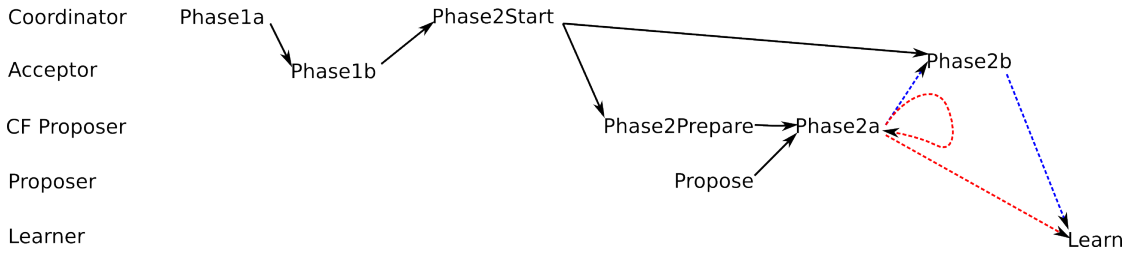

Fig. 1. The flow of actions on good runs; the dashed lines show the two-communication steps to deliver a message. The coordinator starts the round by executing action Phase1a. Acceptors respond to the messages sent in Phase1a on Phase1b. On action Phase2Sart, based on the responses gotten, the coordinator sends a 2S message to collision-fast proposers. On Phase2Prepare, collision-fast proposers get ready to actually propose some value. Regular proposers send proposal requests on phase Propose; if the proposal is from the collision-fast proposer itself, the propose message delivered with no delay. On Phase2a, collision-fast proposers send their proposals to the acceptors. Acceptors accept the proposals and let learners know on Phase2b. Learners perceive the decision of a value on phase Learn.

implies [6]. In Collision-fast Paxos, the set of collision-fast proposers depends on the round. Consequently, the failure of any collision-fast proposer does not prevent learners to learn a decision in two communication steps. When the leader suspects the crash of a collision-fast proposer of the current round, it starts a new round that does not include the failed proposer in the set of collision-fast proposers. 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 IV-A:

- 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.
- 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.

These changes do not affect safety because they do not change the algorithm's variables and make pre-conditions more restrictive only. Besides these changes, we also require that agents keep resending some of their messages in order to overcome transient communication failures.

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 $activep[c]$, and
- $activep[c]$ 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 (cf may be ignored in the comparison without prejudice to the scheme). 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.

V. Atomic Broadcast

To implement atomic broadcast, we use infinitely many M-Consensus instances, each one uniquely identified by a natural number. Atomic broadcast proposers act both as proposers and learners in each of the M-Consensus instances. Algorithm 2 presents Collision-fast Atomic Broadcast in detail.

To broadcast m, a collision-fast proposer p proposes m in the smallest instance of M-Consensus i in which it has neither proposed nor learned anything yet. When a proposer that is

not collision-fast for its current round wants to atomically broadcast a message m, it forwards m to one of the collision-fast proposers of the current round. Since proposers are also learners, they eventually learn the decision of i and can check whether m is in the decision. If not, p re-proposes m in the next free M-Consensus instance.

Assuming a previously agreed 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 must wait until the current instance is complete.

In the normal case, if a collision-fast proposer p fast-proposes a message, then a v-mapping containing such a message is learned in two message steps (see Section IV-A). 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.

Algorithm 2 Collision-fast Atomic Broadcast

I: the set of all Collision-fast Paxos instances used
$CFP(i)!A$: the action or variable A of Collision-fast Paxos instance i

1: $Propose(p, V) \triangleq$
2: $\forall i \in I, CFP(i)!Propose(p, V)$
3: $NewPhase1a(i, c, r) \triangleq$
4: **pre-conditions:**
5: $c = C(r)$
6: $crnd[c] < r$
7: c believes itself to be the leader
8: c heard of a round $r > j > crnd[c]$ for some instance or $CF(crnd[c]) \notin active[c]$
9: **actions:**
10: $CFP(i)!Phase1a(c, r)$
11: $Phase1a(c, r) \triangleq$
12: $\forall i \in I, CFP(i)!NewPhase1a(i, c, r)$
13: $Phase1b(a, r) \triangleq$
14: $\forall i \in I, CFP(i)!Phase1b(a, r)$
15: $Phase2Start(c, r) \triangleq$
16: $\forall i \in I, CFP(i)!Phase2Start(c, r)$
17: $Phase2Prepare(p, r) \triangleq$
18: $\forall i \in I, CFP(i)!Phase2Prepare(p, r)$
19: $Phase2a(p, r, V) \triangleq$
20: **pre-condition:**
21: p has not yet proposed V
22: **action:**
23: LET $i = Min([j : CFP(j)!pval[p] = none])$
24: $CFP(i)!Phase2a(p, r, V)$
25: $Phase2b(i, a, r) \triangleq$
26: $CFP(i)!Phase2b(a, r)$
27: $Learn(i, l) \triangleq$
28: $CFP(i)!Learn(l)$

VI. Related Work

Interactive Consistency [10] is related to M-Consensus in that the proposals of many proposers are included in the decision. More specifically, in interactive consistency each proposer p_i proposes a value v_i and the algorithm must decide on a vector D such that $D[i] = v_i$ if p_i does not crash and $D[i] \in \{v_i, \perp\}$ otherwise. (This property is equivalent to Validity in IC [11].) Since in M-Consensus not all proposed values must be included in the decision, even if no process crashes, M-Consensus is weaker than IC. This explains why M-Consensus can be solved with unreliable failure detectors [3] while IC requires a perfect one [12].

The Non-Blocking Weak Atomic Commitment (NB-WAC) is also related to M-Consensus in that different votes are taken into account to decide on a transaction outcome and failure suspicions may cause votes to be "overrulled" [13]. The similarities are more plainly viewed if the Paxos Commit algorithm used to solve the problem, since it uses a Paxos consensus instance to decide on each vote [14]. Even though NB-WAC is used to decide on either Commit or Abort, the Paxos Commit could easily be changed to decide on some other function of the votes, such as a sequence of the votes, the order in which they should be delivered in an atomic broadcast protocol, similarly to our use of Collision-fast Paxos. Differently from our protocol, however, such a Paxos Commit based atomic broadcast protocol would not be collision fast.

Mencius [15] uses a simplified consensus protocol, solved by a modified version of Paxos, namely Coordinated Paxos. In their protocol, only the coordinator is free to propose a value, including a no-op value; other processes may only propose no-op. This way, whenever the coordinator of a Coordinated Paxos instance proposes no-op, it is guaranteed that no-op will be the decision, and therefore any agent that sees such a proposal may learn the decision in one communication step. Multiple instances are used in parallel, to agree on a set of messages to deliver, in a way similar to what is done by our CFPaxos. In fact, in the absence of failures and failure suspicions, both algorithms have a similar message exchange pattern. Under failures or failure suspicions, CFPaxos requires that the coordinator start a new round, while in Mencius other proposer may propose no-op on behalf of the suspected node. In CFPaxos, if the coordinator fails, a new one must be selected, but the new coordinator can choose a different set of collision-fast proposers for the next round of all CFPaxos instances and, consequently, keep the execution collision-fast. In Mencius, if one of the coordinator fails, the others must keep voting no-op in its instances. Hence, even though the authors provide a method to speed up the no-op voting, strictly speaking, the algorithm is not collision-fast.

There exist other atomic broadcast algorithms that can deliver messages within two steps in some optimistic runs (e.g., [16], [17], [18]), but the only protocol we are aware of that is truly collision-fast is [19]. It tolerates more than a single failure but, instead of relying on consensus, it extends the timestamp-based algorithm presented by Lamport in [1]. In contrast to the approach in [19], 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 [19] when failures happen.

VII. Conclusion

In this paper, we have discussed the implementation of a collision-fast atomic broadcast protocol. Since the traditional approach to implementing atomic broadcast based on standard consensus cannot result in a resilient collision-fast implementation, we have proposed a new agreement problem called M-Consensus that allows multiple proposals to figure in the problem's decision. Our solution 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.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978. [Online]. Available: http://portal.acm.org/citation.cfm?id=359563

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[3] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Communications of the ACM*, vol. 43, no. 2, pp. 225–267, 1996. [Online]. Available: http://www.acm.org/pubs/toc/Abstracts/jacm/226647.html

[4] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: http://portal.acm.org/citation.cfm?id=279227.279229

[5] ——, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, October 2006.

[6] ——, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[7] R. Schmidt, L. Camargos, and F. Pedone, "On collision-fast atomic broadcast," EPFL, Tech. Rep., 2007. [Online]. Available: http://infoscience.epfl.ch/getfile.py?recid=100857

[8] L. Lamport, "Generalized consensus and paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, 2004.

[9] M. J. Fischer, N. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[10] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980. [Online]. Available: http://doi.acm.org/10.1145/322186.322188

[11] M. Raynal, "A short introduction to failure detectors for asynchronous distributed systems," *SIGACT News*, vol. 36, no. 1, pp. 53–70, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1052796.1052806

[12] J.-M. Hélary, M. Hurfin, A. Mostefaoui, M. Raynal, and F. Tronel, "Computing global functions in asynchronous distributed systems with perfect failure detectors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 9, pp. 897–909, 2000.

[13] R. Guerraoui, "Revisiting the relationship between non-blocking atomic commitment and consensus," in *Distributed Algorithms*. Springer, 1995, pp. 87–100.

[14] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.

[15] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 369–384.

[16] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, no. 2, pp. 97–107, April 2002. [Online]. Available: http://dx.doi.org/10.1007/s004460100061

[17] ——, "Optimistic atomic broadcast: a pragmatic viewpoint," *Theoretical Computer Science*, vol. 291, no. 1, pp. 79–101, January 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=795644

[18] 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)*, Osaka, Japan, Oct. 2002, pp. 92–101.

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