Replicated State Machines for Collision-Prone Wireless Networks

Gregory Chockler Seth Gilbert

MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA 02139
{grishac,sethg}@theory.csail.mit.edu

Abstract

Replicated state machines have long been used as a means to implement fault-tolerant services in fault-prone distributed networks. In this paper, we extend the replicated state machine paradigm to wireless, ad hoc networks that are prone to collisions, message loss, and crash failures. Our algorithm is designed specifically for ad hoc deployments, where the participants are *a priori* unknown, and is adaptive to varying numbers of participants. Finally, our algorithm is efficient in the sense that the protocol messages are constant-sized, and the state machine returns responses in constant time.

Contact author:	Seth Gilbert
Address:	MIT CSAIL, the Stata Center
	32 Vassar St. (32-G694)
	Cambridge, MA 02139
E-mail:	sethg@mit.edu
Phone:	617-253-7583
Track:	Regular
Eligible student paper:	Yes (Seth Gilbert is a student.)

1 Introduction

Recently, there has been an increased interest in wireless ad hoc networks which, due to their ability to bring computation to environments with minimal infrastructure, are widely considered to be the next generation computing platform. However, developing applications for these environments is quite challenging. One of the primary concerns faced by system developers is the increased vulnerability to failures resulting from the fragility and power limitations of wireless devices. These challenges call for a systematic approach to designing fault tolerant services. In this paper, we focus on one such approach, the *Replicated State Machine (RSM)* [14, 21] methodology, where a reliable service is constructed from a collection of fault-prone replicas.

Over the years, the RSM methodology has evolved into a set of generic, commonly-used implementation techniques such as the timestamp-based protocol of Lamport [14], Paxos [15], group communication-based protocols [13, 1], and many others. However, the existing implementations were designed for traditional "wired" networks (especially, local area networks), and as a result, do not properly address several important concerns arising in wireless settings (see Section 2 for a detailed discussion). In this paper, we extend the RSM paradigm to collision-prone wireless networks by exhibiting an algorithm that, to the best of our knowledge, is the first generic technique for implementing a reliable service in these emerging environments. We focus on implementing an RSM in short range networks where the participating nodes are located within communication range of each other.

Implementing the RSM paradigm in wireless networks confronts many challenges. First, communication in these networks is unreliable: collisions and other electromagnetic interference can cause significant message loss. Furthermore, due to unknown deployment densities, the transmission schedules should be carefully controlled so as not to overwhelm the medium with too many collisions. Another problem introduced by ad hoc deployment is that the total number of participating nodes may be unknown, and there may be no *a priori* bound on the number of failures.

We address these challenges by taking advantage of the following physical characteristics of single-hop wireless networks. First, message broadcast within a short range is nearly instantaneous: if a non-failing node broadcasts a message, then unless a collision occurs, the message is received by all nearby nodes in a timely fashion. Furthermore, as indicated by recent work [23, 20, 6], collisions can often be detected with a reasonable precision.

We capture these properties using a synchronous round model: if a message is broadcast in a round, then each nearby node either receives the message or detects a collision. We model the collision detection guarantees using the approach introduced in [5]. In particular, for most of this paper, we assume that the collision detection is complete (i.e., able to detect actual collisions) and eventually accurate (i.e., eventually capable of not mistaking noise for a collision). We also discuss (see Section 7) how our algorithms can be extended for weaker classes of collision detectors.

In this model, we present a generic technique for emulating a reliable service using the RSM approach. The emulation is structured as a synchronous state machine in which each round consists of three parts: receiving inputs, updating the state, and producing outputs. Sometimes input and output messages can be lost due to collisions incurred by the communication medium. In these cases, our protocol guarantees that the state machine detects a collision. An alternative would be to attempt to recover the lost messages. However, in the presence of collisions, it is impossible to guarantee that the lost messages are recovered within a bounded number of rounds. Therefore, we opt for a simpler and more efficient implementation while providing collision detection as a fallback option.

The emulation algorithm employs a special type of a contention manager (see, e.g., [5, 12]), called a *wake-up service*. The wake-up service designates nodes as active or passive and guarantees that eventually the overall number of active nodes is small enough to ensure collision-free communication. Our wake-

up service differs from leader election primitives commonly used by RSM implementations as it does not guarantee a unique active node. In practice, contention managers are typically implemented using backoff protocols.

Contributions. In this paper, we present an algorithm that emulates a synchronous, collision-aware state machine with the following properties:

• *Collision Tolerance:* Our protocol preserves safety at all times, even when the overall number of active nodes is not small enough to guarantee collision freedom.

• *Adaptivity:* Our protocol uses the contention manager to respond adaptively to changing numbers of participants. It guarantees termination once the number of active nodes is sufficiently small to guarantee collision freedom.

• *Efficiency:* Our protocol is efficient in the sense that it requires only a constant number of communication rounds to implement a single state-machine round. Moreover, the protocol messages are all of constant size.

• *Anonymity:* Our protocol is designed to cope with ad hoc deployments: it does not require any knowledge of the number of participants, nor does it require that the participants have unique identifiers.

• *Failures, joins and leaves:* Our protocol can tolerate an arbitrary number of node crash failures and leaves. Handling joins is simple due to the protocol's ability to cope with anonymity. For the sake of presentation, joins are omitted from the basic description, and are described as an extension in Section 7.

The rest of this paper is organized as follows: In Section 2, we discuss the related work. Our system model is described in Section 3, and we introduce the collision-aware state machine in Section 4. The implementation appears in Section 5 and a sketch of the proof in Section 6. Section 7 discusses extensions and optimizations of the basic protocol.

2 Related Work

The concept of implementing fault-tolerant services using replicated state machines was first introduced by Lamport in [14] and popularized by Schneider in [21]. The original RSM paper [14] also presents a technique for implementing a fault-tolerant state machine in a message-passing system with fail-stop replicas (i.e., with reliable failure detection). For more general networks, the Paxos protocol and its variants [15, 16, 17] provide an RSM implementation resilient to $n/2$ replica crashes, where n is the overall number of replicas, and tolerates arbitrarily long periods of instability during which time bounds can be violated and messages can be lost. Castro and Liskov present a Byzantine resilient version of the Paxos protocol [3]. A separate line of research was dedicated to implementing RSMs on top of view-oriented group communication systems (e.g., [13, 1, 10]).

All of the above protocols were targeted to traditional "wired" networks. As a result, several important concerns arising in wireless ad hoc environments are not reflected in their design. In particular, they lack the ability to dynamically adapt their patterns of communication to the number of participating nodes. This could lead to situations where the number of simultaneously broadcasting nodes is too high for the underlying network to handle (at least within a reasonable time). As a result, collisions might never cease to occur (or it might take arbitrarily long to achieve collision freedom), thus precluding these protocols from ever making progress, or terminating in bounded time. The above techniques are also not suitable for anonymous settings as they rely on *a priori* knowledge of the number of participants and their identities, and may also require the nodes to have unique identifiers. An algorithm for atomic broadcast that tolerates dynamic networks with an unknown set of participants is presented by Bar-Joseph et al. [2]. Although their protocol can be used to implement a reliable state machine, it is nevertheless inapplicable in our setting as it assumes reliable, collision-free communication.

The only RSM implementations we are aware of that were explicitly targeted to wireless ad hoc networks are [9, 8, 7]. The protocols described in these papers assume a synchronous environment with collision-free communication, and thus again are not applicable in our collision-prone setting. The protocols described in these papers tolerate any number of crash failures; some are capable of recovering from arbitrary state corruption (viz., self-stabilization).

Our synchronous system model with collision-prone broadcast, contention management, and collision detection was introduced in [5, 4]. In particular, [5] presents a formal framework for specifying collision detection properties based on *completeness* (i.e., the ability to detect collisions when they occur) and *accuracy* (i.e., the ability not to generate false positives) and studies their computational power based on the ability to solve consensus. The consensus algorithms of [5] can potentially be used to implement a state machine in a straightforward fashion, for example, by repeatedly solving consensus in each state-machine round. Since these consensus algorithms only terminate after the communication medium and the collision detectors stabilize, they cannot be directly used to achieve fixed-length state-machine rounds throughout the entire execution.

3 The System Model

We consider a single-hop wireless broadcast network consisting of a fixed but *a priori* unknown collection of nodes $P = \{p_1, p_2, \dots\}$ where all nodes are located within communication range of each other. The number of nodes is unknown, and nodes may not have unique identifiers. Nodes can fail by crashing at any point during the execution of the algorithm.

Nodes communicate by broadcasting and receiving messages. The broadcast communication satisfies basic integrity and no-duplication properties: every message received was previously broadcast, and each message is received at most once. Due to the nature of wireless communication, a broadcast is atomic.

We assume that the system is synchronous: both the nodes' clock skews and the inter-node communication delay are bounded by known constants. For simplicity, we assume that the processing is divided into synchronous rounds. In each round, each node may: (1) broadcast a message, (2) receive a subset of messages that were broadcast, and (3) update its state based on its current state and the received messages.

The communication medium is prone to *collisions*. As a result of a collision, each node can loose an arbitrary subset of messages that were broadcast in a round. Moreover, collisions may affect nodes in a non-uniform way; when a node broadcasts a message, some nodes may receive the message while others may not.

We also assume the existence of a special type of contention manager, a *wake-up service* which can be queried by a node to determine whether it should be active (i.e., whether it should broadcast) in a round. The goal of the wake-up service is to reduce the number of active nodes. The network guarantees that if only a small number of nodes broadcast, then messages are delivered without collisions. Combining these two properties together, we stipulate the following:

Property 1 (Eventual Collision Freedom). *There exists a round, r_{ecf}, such that in any round $r \geq r_{ecf}$, if only* active *nodes broadcast in round r, then no collisions occur in round r.*

The wake-up service can be easily implemented using a simple backoff protocol (see, e.g., [18, 11, 22, 19].)

We assume that the MAC layer of every node $p_i \in P$ is augmented with a *collision detector*. We say that p_i *detects* a collision in round r when it receives a collision notification, $\pm$. Collision notifications indicate that some message might have been lost; they do not provide any information with respect to the number of lost messages or the identities of their senders. We assume that the collision detectors are in the class $\Diamond\mathcal{AC}$, as defined in [5], meaning they satisfy the following properties:

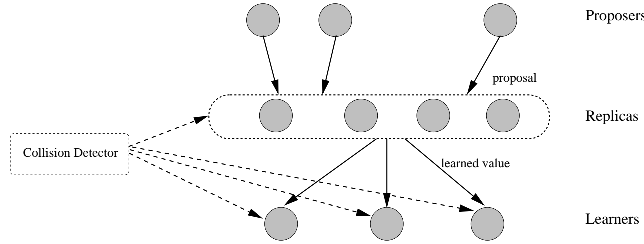

Figure 1: High-level diagram of a replicated state machine consisting of proposers, replicas, and learners.

Property 2 (Eventual Accuracy). *For each execution, there exists a round r_{acc} such that for each round $r' \geq r_{acc}$, and each node $p_i \in P$: if p_i detects a collision in r', then p_i does not receive some messages that were broadcast in r'.*

Property 3 (Completeness). *For every round r of each execution, if p_i does not receive some messages that were broadcast in r, then p_i detects a collision in r.*

The safety of our implementation depends on Property 3; liveness depends on Properties 1 and 2.

4 Collision-Aware State Machines

In this section, we show how to modify the well-known state machine abstraction for use in collision-prone environments. To this end, we introduce the notion of a *collision-aware* state machine, that is, a state machine that can detect collisions when either inputs or outputs are lost.

Formally, a collision-aware state machine is an automaton which is a 5-tuple consisting of: (i) a set of legal states, V, (ii) an initial state, v_0, (iii) a set of proposals, $inputs$, (iii) a set of values to learn, $outputs$ and (iv) a transition function, δ, that maps from $V \times \mathcal{P}(inputs) \cup \{\pm\}$ to $V \times outputs$. Notice that the transition function, δ, can update the state as a result of a collision.

Each participating node is assigned at least one of three possible roles: *proposer*, *replica*, or *learner*[1]. The state machine operates in a round-based fashion. Each state-machine round consists of three parts: (i) proposers send proposals, $p \in inputs$, to the state machine, (ii) the replicas receive the proposals, update the state and broadcast output values, and (iii) learners receive the output values, $\ell \in outputs \cup \{\pm\}$, from the state machine and output them. The components of a collision-aware state machine are depicted in Figure 1.

Formally, we say that an algorithm A implements a state machine if the resulting state-machine rounds are consistent with a valid execution of the automaton. That is, for each state-machine round r, there is a state $v_r \in V$ and a set of proposals, $Q_r \subseteq inputs \cup \{\pm\}$ that satisfy the following properties: (i) The sequence of states is consistent with the transition function, that is, $v_{r+1} = \delta(v_r, Q_r).state$. (ii) The learned values are consistent with the transition function, that is, if $\ell \neq \pm$ is learned in state-machine round r, then $\ell = \delta(v_r, Q_r).out\text{-}msg$. (iii) Each Q_r is a subset of the proposals in state-machine round r; if some proposal from round r is $\notin Q_r$, then the state machine detects a collision, i.e., $\pm \in Q_r$. (iv) If the state machine has failed in state-machine round r, then no value is learned in or after round r.

So far our defi nition captures only the safety properties. We also require that: (v) Each state-machine round is implemented by a constant number of communication rounds. (vi) After eventual collision freedom (i.e., after round r_{ecf}) and after eventual accuracy (after round r_{acc}) then the state machine experiences no further collisions. (vii) If at least one replica does not fail, then the state machine does not fail.

[1]This terminology was inspired by Lamport [16]. We use the term replica, instead of acceptor, to emphasize the replication.

ballot	veto-1	veto-2	Replica	Learner
$\surd$	$\surd$	$\surd$	Green	Green
$\surd$	$\surd$	X	Yellow	Red
$\surd$	X	X	Orange	Red
X	X	X	Red	Red

Figure 2: Table indicating how a replica or learner responds to messages in the ballot/veto-1/veto-2 rounds. A $\surd$ indicates that the replica receives a message in that round, and no collisions or veto indications are received. An X indicates that the replica/learner does not correctly receive the message in that round.

Notice that our definition of a replicated state machine is slightly more general than is usual. Typically, a state machine accepts operation *requests* and produces *responses*. We allow the state machine to produce an output independent of prior requests, and likewise a proposal does not necessitate a response.

5 Algorithm Description

In this section we present the basic replicated state machine algorithm. The interface and data structures of the protocol are presented in Figure 4, and the code is presented in Figure 5.

The protocol for the proposer is simple: during the first phase of a state-machine round, the propose phase, the proposers broadcast their proposals. The protocol at the replicas and learners is somewhat more involved. Typically, a protocol to implement a replicated state machine (e.g., [15]) uses repeated instantiations of consensus to update the state, ensuring that all nodes transition to the same next state and that the next state is valid. Unfortunately, as was shown in [5], it is impossible in the wireless setting to achieve consensus in a constant number of communication rounds prior to the onset of stability (i.e., prior to round r_{ecf} and round r_{acc}).

The first insight is that the requirements at the replicas are different than the requirements at the learners. The replicas require agreement, as is typical. However, the behavior of the replicas is not externally observable, and as a result, they are not required to produce an output in every state-machine round. By contrast, the learners must output a decision in every state-machine round, but require only "weak" agreement: if two learners choose to output two different values in the same state-machine round, then one of them must output a collision, $\pm$. Learners (unlike replicas) can disagree, but only in that some may detect a collision instead of the selected value.

The Learner Protocol. Therefore the learner's protocol is relatively simple. The learners listen to the messages sent by the replicas; if a learner detects a collision, indicating some uncertainty with the protocol, it outputs $\pm$, which is always safe. On the other hand, all the learners that detect no collisions receive the same set of messages, by Property 3. Therefore, these learners can choose a consistent output, as required.

The Replica Protocol. The replica protocol must provide the traditional guarantees of consensus, while maintaining constant-sized messages and allowing the learners to output consistent values in every state-machine round. If the replicas simply begin a new instance of consensus in every round, for example using the consensus protocol presented in [5], then the messages might become too large: they might need to carry data pertaining to an arbitrary number of concurrent instances of consensus. Instead, we use an approach in which each new instance of "consensus" effectively supersedes all previous instances. The resulting protocol is reminiscent of three-phase commit. The goal of the replicas is to determine which state-machine rounds to accept (analogous to a commit), and which to reject (analogous to an abort). This requires two communication rounds, one to prepare the replicas, and a second to commit.

There are therefore three phases which follow the first phase, the propose phase: a ballot phase in which active replicas broadcast a ballot, and then two veto phases. When a replica detects a collision in the ballot

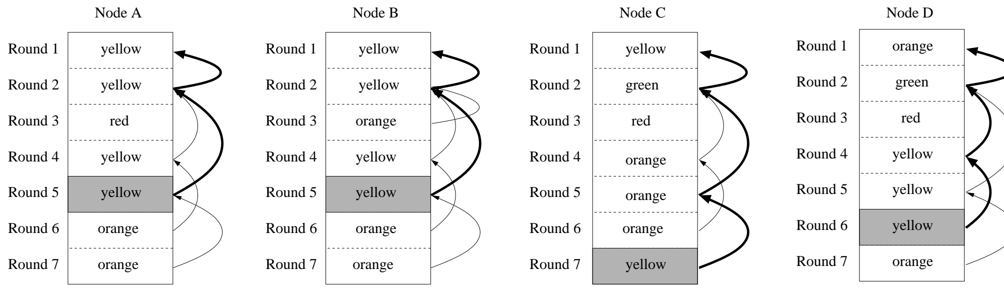

Figure 3: An example of four ballot histories. The arrows represent *tentative-round* pointers, and are stored in the *ballot* array. The round colors are stored in the *round-status* array.

phase, it broadcasts a veto in the veto-1 phase; when a replica detects a collision or a veto in the veto-1 phase, it broadcasts a veto in the veto-2 phase. Each replica assigns a color to each state-machine round, based on whether the ballot is vetoed, and if so, in which round the veto occurs. See Figure 2 for a table indicating how the colors are assigned. If a replica determines that a state-machine round is green or yellow, it chooses to *accept* the state-machine round; otherwise, if the state-machine round is orange or red, the replica chooses to *reject* the state-machine round.

This 3-phase-commit-like protocol allows the replicas to reconstruct a history of the state machine, based on the result of accepting or rejecting a state-machine round. The *ballot* includes a pointer to the last round that a replica has chosen to accept (i.e., the last round that was designated yellow or green). The protocol ensures that two replicas can only differ in their color assignment by one shade. For example, if one replica assign the color yellow to a state-machine round, then every other replica assigns either orange or green to that state-machine round. If a replica designates a round as green, then it can be sure that every other replica has chosen to accept that round, and no future ballot will choose to abort the round. The converse holds in the case where a round is rejected. Therefore by examining the backward pointers in the ballots, a replica can determine the history of the state machine.

An Example. An example appears in Figure 3. The shaded round (e.g., round 5 for node A) represents the last round that a replica chose to accept; all later rounds (e.g., rounds 6 and 7 for node A) are either orange or red. The arrows indicate the rounds referenced in each ballot. For example, the ballot in round 2 referenced round 1 (from which we can conclude that node D was not the smallest active replica in round 2). Different replicas may have differing beliefs about the history of the state machine, depending on the colorings of the rounds. The darker arrows indicate each node's belief about the history. For example, node C, which colors round 7 yellow, prefers a different history to node D, which colors round 7 orange. However, green rounds synchronize the beliefs of the nodes. Notice in the example that every node has accepted rounds 1 and 2.

The Complete Algorithm. The complete algorithm, presented in Figure 5, consists of four phases:

- propose: The proposers send out proposals. The replicas receive the proposals and assemble a ballot.
- ballot: The active replicas broadcast the ballot. The replicas and learners store the ballot in their ballot log. If a replica or learner detects a collision, it colors the state-machine round red.
- veto-1: Replicas broadcast vetoes if they did not receive the ballot. If a replica or learner then detects a collision or receives a veto, it colors the state-machine round orange. Otherwise, if the state-machine round is not red or orange, the replicas choose to accept the state-machine round and update the *tentative-round* and *tentative-state* by walking backwards through the ballot log (see Figure 5, **function** update-tentative).
- veto-2: Replicas broadcast vetoes if they detected a collision or received a veto message in the first veto phase. If a replica or learner then detects a collision or receives a veto, it colors the state-machine round

Figure 4: Replicated State Machine — Interface and Data Structures

```
1  Proposer Interface:
2    Input propose(), returns client's proposal p ∈ inputs
3    Input recv(), returns incoming messages
4    Output bcast(outgoing-msg), broadcasts outgoing-msg
5    Input fail(), returns whether proposer has failed
6
7  Replica Interface:
8    Input recv(), returns incoming messages
9    Output bcast(outgoing-msg), broadcasts outgoing-msg
10   Input Wakeup(), returns whether replica is active
11   Input fail(), returns whether replica has failed
12
13 Learner Interface:
14   Input learn(ℓ), notifies client of value ℓ ∈ outputs
15   Input recv(), returns incoming messages
16   Output bcast(outgoing-msg), broadcasts outgoing-msg
17   Input fail(), returns whether learner has failed

19 Data Structures:
20   role, initially proposer, replica, or learner
21   state, tentative-state ← v0, the initial state
22   round, last-good-round, tentatve-round ← 0
23   round-status[ ] ← ⟨green, green, ..., green⟩
24   ballots[ ], an array of:
25     tentative-round, initially 0
26     out-msg, initially ⊥
27     proposals, initially ∅
28   phase ← propose
29   incoming-msgs, outgoing-msg ← ⊥
30   failed ← false
```

yellow. If the state-machine round is still green, then replicas and learners can commit to accepting this state-machine round. In this case, replicas update their $last\text{-}good\text{-}round$ and $state$, and learners output the appropriate $out\text{-}msg$. Otherwise, the learners simply output collision, $\pm$.

6 Proof of Correctness

In this section we sketch a proof that the algorithm correctly implements a synchronous, collision-aware state machine. Fix an execution of the state machine emulator. For each replica and learner, i, we construct a tree representing the set of possible state-machine executions that are consistent with the local actions taken by node i. Each node in the tree represents a state, and each edge represents a transition. Initially, the tree consists of a single node, v_0. After each state-machine round, r, the tree is updated:

- If node i has failed prior to the end of state-machine round r, then the tree is left unchanged.
- If $round\text{-}status[r]_i$ is red, then a "collision node" is added to every leaf of i's tree. If s is the label of a leaf, then $\delta(s, \pm).state$ is the label of the collision node. The new edge is labeled $\pm$.
- If $round\text{-}status[r]_i$ is not red, then:
  - Add a new "collision node" to every leaf of i's tree. If s is the label of a leaf, then $\delta(s, \pm).state$ is the label of the collision node. The new edge is labeled $\pm$.
  - Let r' be the $tentative\text{-}round$ included in the ballot from round r. As we will show, some non-"collision node" must have been added to the tree in r'. Begin at that node, and follow $\pm$ edges until a leaf is reached. Let s be the label of this leaf, and m be the $incoming\text{-}msgs$ included in the ballot for round r. Add a new non-collision node child to this leaf with label $\delta(s, m).state$; the label of the new edge is m.
- If $round\text{-}status[r]_i$ is green, then prune node i's tree: remove all nodes in the tree not on a root-to-leaf path ending at the newly added non-collision node.

If there exists a root-to-leaf path in every node's tree, then that path demonstrates a valid execution of the state machine. The key lemma, which follows from the use of veto rounds, is that:

Lemma 6.1. *For all r, the color $round\text{-}status[r]$ at any two nodes i and j differ by at most one shade.*

The following lemma, then, indicates the existence of a root-to-leaf path in every node's tree, and at the same time shows that the (inductive) tree construction is feasible:

Figure 5: Replicated State Machine Implementation

```
33  Round r Broadcast:
34    if fail() then failed ← true
35    outgoing-msg ← ⊥
36    begin cases:
37      case phase = propose:  round ← round + 1
38                             if (role = proposer) then outgoing-msg ← propose()
39      case phase = ballot:   if (role = replica) and (Wakeup()) then outgoing-msg ← ballots[round]
40      case phase = veto-1:   if (role = replica) and (round-status[round] = red) then outgoing-message ← veto
41      case phase = veto-2:   if (role = replica) and (round-status[round] ∈ {red,orange}) then outgoing-msg ← veto
42    end cases
43    if ¬ failed then bcast(outgoing-msg)
44
45  Round r Receive:
46    incoming-msgs ← recv()
47    begin cases:
48      case phase = propose:  if (role = replica) then
49                               ballots[round] ← ⟨tentative-round, δ(tentative-state, incoming-msgs).out-msg, incoming-msgs⟩
50
51      case phase = ballot:   if (role = replica or learner) then
52                               if (±∈ incoming-msgs) or (incoming-msgs = ∅) then
53                                 round-status[round] ← red
54                               else
55                                 ballots[round] ← min(incoming-msgs)
56
57      case phase = veto-1:   if (role = replica or learner) then
58                               if (veto ∈ incoming-msgs) or (±∈ incoming-msgs) then
59                                 round-status[round] ← min(orange, round-status[round])
60                               if (role = replica) and (round-status[round] ∉ {red,orange} then
61                                 ⟨tentative-state, tentative-round⟩ ← update-tentative(round, ballots, state, last-good-round)
62
63      case phase = veto-2:   if (role = replica or learner) then
64                               if (veto ∈ incoming-msgs) or (±∈ incoming-msgs) then
65                                 round-status[round] ← min(yellow, round-status[round])
66                               else if (round-status[round] = green) then
67                                 if (role = learner) and (¬ failed) then learn(ballots[round].out-msg)
68                                 if (role = replica) then
69                                   last-good-round ← tentative-round
70                                   state ← tentative-state
71                               else if (role = learner) and (¬ failed) then learn(±)
72                               phase ← phase + 1
73    end cases
74    phase ← phase + 1
75
76  function update-tentative(temp-round, temp-ballots, temp-state, temp-last-good-round)
77    next-good ← temp-round
78    prev-good ← temp-ballots[temp-round].tentative-round
79    tentative-status[temp-round] ← green
80    while prev-good ≥ max(1,temp-last-good-round) do
81      for i = (prev-good + 1) up to (next-good -1) do tentative-status[i] ← red
82      tentative-status[prev-good] ← green
83      next-good ← prev-good
84      prev-good ← temp-ballots[next-good].tentative-round
85    for i = temp-last-good-round to temp-round do
86      if (tentative-status[i] = green) then
87        in-msgs ← temp-ballots[i].in-msgs
88        temp-state ← δ(temp-state, in-msgs)
89      else if (tentative-status[i] = red) then
90        temp-state ← δ(temp-state, ±)
91    return ⟨temp-state, temp-round⟩
92
```

Lemma 6.2. *Assume that at least one replica does not fail, and consider the set of trees after executing r rounds of the state machine. Then there is a root-to-leaf path in every non-faulty node's tree. Moreover, if at any point in the execution, $tentative\text{-}round_j = r'$ for some node j and some $r' < r$, then at that point in the execution every tree contains (i) a non-collision node at level r' and (ii) a path from that non-collision node to a level r leaf following only collision edges.*

Proof (sketch). We prove this lemma by induction on the number of rounds. Initially the result is trivially satisfied, since every tree consists of a single node labeled v_0.

Consider how the tree of a replica or learner, i, is updated after round r of the state machine. If $round\text{-}status[r]_i$ is red or orange, then no other replica or learner has $round\text{-}status[r]$ of green. Therefore every replica and learner adds a collision node to its tree, thus extending the root-to-leaf path inductively assumed in the set of trees from after $r - 1$ rounds of the state machine.

If $round\text{-}status[r]_i$ is green, then every other replica or learner has $round\text{-}status[r]$ of yellow or green. Therefore every replica and learner adds a non-collision node, extending the root-to-leaf path found in every tree after $r - 1$ rounds (even while potentially pruning other nodes).

Finally, if $round\text{-}status[r]_i$ is yellow, then there are two cases: either every other replica or learner has $round\text{-}status[r]$ orange, or every other replica or learner has $round\text{-}status[r]$ yellow. In the first case, every replica and learner adds a collision node; in the second case every replica and learner adds a non-collision node. In both cases, the root-to-leaf path is extended.

The final property to ensure is related to $tentative\text{-}round$. If some node has $tentative\text{-}round$ set to r', then it must have $round\text{-}status[r']$ equal to either green or yellow. Moreover, every round since r' must have been orange or red; otherwise the $tentative\text{-}round$ would have been updated. Therefore no replica or learner could have had a green round since r', and therefore the non-collision node from r', and the collision nodes since r', remain in every node's tree. □

We conclude that the emulated state machine is correct. The nodes on the shared root-to-leaf path correspond to states, v_r, and the edges correspond to inputs, Q_r. The tree construction ensures that this is a valid execution. The outputs are consistent since learners only produce outputs in green rounds. The state machine fails only if all the acceptors fail, and in this case it produces no further outputs. Each state-machine round requires only a constant number of communication rounds, and once collisions cease and the collision-detector becomes accurate, all rounds are green and no more collisions are reported. We thus conclude:

Theorem 6.3. *Every execution of the algorithm emulates a synchronous, collision-aware state machine.*

7 Extensions and Optimizations

In this section we discuss some further extensions and optimizations to the basic algorithm.

Dynamic Networks. The only modification required to handle joining nodes is a mechanism to transfer the state from the old replicas to the new replicas. This can be accomplished by adding an extra two phases: join, in which new replicas request the current state, and join-ack, in which active replicas send a response.

Unreliable Collision Detectors. In [5], we present several classes of weaker collision detectors. The most interesting of these is maj-$\Diamond\mathcal{AC}$: *majority-complete*, *eventually accurate* collision detectors that report a collision if a majority of messages are not received in a round. (Contrast this to complete collision detectors, $\Diamond\mathcal{AC}$, where a collision is reported if even one message is lost.) These weaker collision detectors are much more feasible to implement in real wireless networks. In order to tolerate the weaker collision detectors, the following changes to the protocol are needed. First, a pre-ballot phase is added immediately prior to the ballot phase in which active replicas circulate their ballots. All the replicas then adopt the ballot with the

minimal identifi er. In the ballot phase, the active replicas broadcast the revised ballot. Replicas veto if all the received ballots are not the same. If no vetoes occur, then every replica received a majority of the messages sent in the ballot phase, and each received a unique ballot. Since all majorities intersect, this ensures that every replica in fact received the same ballot. The rest of the protocol remains as before.

Message Overhead. Notice that every replica receives the proposals from the proposers, or detects a collision, in the propose phase. Therefore there is no reason to include these proposals in the ballot. If a replica detects a collision during the propose phase, it designated the round as red, and the protocol continues as before. With this optimization, the per-round message overhead for a node (compared to simply executing the state machine on a single, unreliable node) is simply an additive constant number of bits (i.e., the *tentative-round* fi eld of the ballot, and any veto messages).

References

[1] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Hebrew University, 1995.

[2] Z. Bar-Joseph, I. Keidar, N. Lynch. Early-delivery dynamic atomic broadcast. *DISC*'02.

[3] M. Castro, B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[4] G. Chockler, M. Demirbas, S. Gilbert, N. Lynch, C. Newport, T. Nolte. Reconciling the theory and practice of unreliable wireless broadcast. *International Workshop on Assurance in Distributed Systems and Networks (ADSN)*, 2005. To appear.

[5] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, T. Nolte. Consensus and collision detectors in wireless ad hoc networks. *PODC*'05. To appear.

[6] J. Deng, P. Varshney, Z. Haas. A new backoff algorithm for the IEEE 802.11 distributed coordination function. *Communication Networks and Distributed Systems Modeling and Simulation*'04.

[7] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, T. Nolte. Virtual stationary automata for mobile networks. Tech report, MIT, 2005.

[8] S. Dolev, S. Gilbert, N. Lynch, E. Schiller, A. Shvartsman, J. Welch. Virtual mobile nodes for mobile ad hoc networks. *DISC*'04.

[9] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *DISC*'03.

[10] A. Fekete, N. Lynch, A. Shvartsman. Specifying and using a partitionable group communication service. *PODC*'97.

[11] R. Gallager. A perspective on multiaccess channels. *IEEE Trans. Information Theory*, IT-31:124–142, 1985.

[12] M. Herlihy, V. Luchangco, M. Moir, I. William N. Scherer. Software transactional memory for dynamic-sized data structures. *PODC*'03.

[13] I. Keidar, D. Dolev. Broadcast in the face of network partitions: Exploiting group communication for replication in partitionable networks. D. Avresky Ed., *Dependable Network Computing*, chapter 3. Kluwer Academic Publications, 2000.

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

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

[16] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[17] B. Lampson. How to build a highly available system using consensus. *WDAG*'96.

[18] K. Nakano, S. Olariu. Uniform leader election protocols in radio networks. *ICPP*'01.

[19] K. Nakano, S. Olariu. A survey on leader election protocols for radio networks. *ISPAN*'02.

[20] J. Polastre, D. Culler. Versatile low power media access for wireless sensor networks. *Embedded Networked Sensor Systems*'04.

[21] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[22] D. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal of Computing*, 15(2):468–477, 1986.

[23] A. Woo, K. Whitehouse, F. Jiang, J. Polastre, D. Culler. The shadowing phenomenon: implications of receiving during a collision. *Technical Report UCB//CSD-04-1313, UC Berkeley*, 2004.