

## 1 Introduction

This class period focused on a theoretical application of collision-detectors to solve the consensus problem; and the implementation of replicated state machines, to produce fault-tolerant wireless networks. We reviewed two papers: “Consensus and Collision Detectors in Wireless Ad Hoc Networks” by G. Chockler, M. Demirbas and S. Gilbert; and “Replicated State Machines” by G. Chockler and S. Gilbert.

## 2 The consensus problem

In most of the programs we write, and in most of the programs we execute from our desktop computers, we assume that the code will be executed properly. Maybe the program doesn’t do what we want it to, but that is the result of buggy code, not correct code that for some reason executes improperly.

However, when the stakes are high—for example, if correct execution of a computer program literally makes the difference between life and death—it is imperative to build a system that will execute correctly despite software or hardware faults. Medical and aviation applications are two areas that require a high level of fault-tolerance. The “consensus problem” is a mathematization of one key aspect of fault tolerance: ensuring that a system of processors makes the correct decision even if one or more processors or links has failed.

Consider an airplane. If the plane has only one altimeter, and that altimeter suddenly stops working correctly, there will be no way for the pilot to know the height at which the airplane is flying. The solution is to add redundancy: equip the airplane with several altimeters and use a method by which the altimeters report their individual readings and decide on a unique value to report to the pilot. Similarly, a layer of redundancy is required for the control of the flaps. The pilot might give the command to raise the flaps on the wings. If there is a sole control for this, and the control is unresponsive (or, worse, interprets the command RAISE as LOWER due to some glitch), the plane might crash. The solution is for multiple controls to receive the RAISE instruction, and mutually decide that the correct instruction is RAISE, even if one or more processors are saying the opposite, or nothing.

Those are applications of the resolution of the consensus problem. Multiple processors had to agree, despite possibly having conflicting start states. Informally, the consensus problem is the following: *the processors in the network have to agree on an output, even though the inputs to each processor may be arbitrary.*

## 2.1 Formal definition

One example of a formal consensus problem is as follows.

*Each processor starts with an input, either 0 or 1. At the end of the execution of the algorithm, all processors output the same number (either 0 or 1).*

A trivial way to achieve that objective would be to run an algorithm that simply requires all processors to output 0 (or 1) no matter what. So the problem statement includes a *Nontriviality Condition*: There exists some input state such that the outcome is 0, and there exists some state such that the outcome is 1.

## 2.2 The context of wireless networks

In the context of ad hoc and wireless networks, we need a “fault-tolerant” consensus algorithm, which is robust in the face of processor and link failures that are common in a setting where nodes are mobile, and have limited battery power and transmission range. Perhaps most importantly, there is no central “brain” or station, so failures have to be resolved by local brains, communicating local information to one another.

Transmission collision is a serious challenge for wireless networks. Figure 1 shows graphically that the messages in wireless networks backlog quickly, as the result of multiple collisions. In fact, experimental studies show that message loss from collisions runs from 20% to 50% of all transmissions!

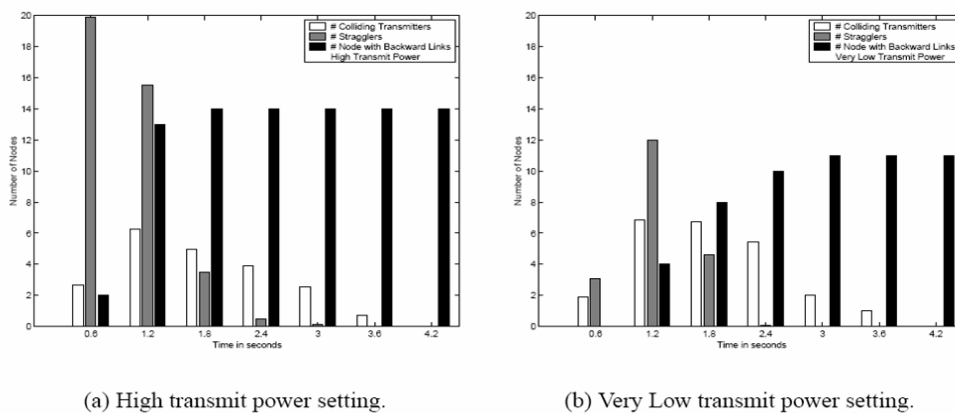


Figure 1: Histogram of experimental data from a network of 169 nodes. White bars are collisions, gray bars are “straggler” nodes that transmit late, and black bars are nodes with delayed packets waiting to transmit. X-axis is time in seconds; Y-axis is # of nodes. Source: “Complex Behavior at Scale: An Experimental Study of Low-Power Wireless Sensor Networks” by D. Ganesan et al.

## 3 Link failure and processor failure

The two main faults that can occur in wireless networks are: (1) communication links go down, and (2) a node or nodes stop working properly. These failures can either be failures of omission—for example, a processor runs out of power and stops participating in the network—or so-called “Byzantine” failures, where the processors can behave arbitrarily or maliciously. It is more difficult computationally to tolerate Byzantine failures than it is to tolerate failures of omission.

### 3.1 Link failure

Simply put, link failure means that the medium crashes so processors cannot communicate. In wireless networks, this can easily result from collisions. Link failure is very hard to compensate for, algorithmically. Even in the case of an extremely simple network graph—two nodes whose internal clocks are completely synchronous with each other—the theoretical lower bound for those nodes to resolve link failure through consensus is as follows:

1. The problem is *unsolvable* by a deterministic algorithm.
2. Any probabilistic algorithm has a probability of node disagreement of at least  $1/(r + 1)$ , where  $r$  is the number of rounds the algorithm takes to execute.

### 3.2 Processor failure

Fault-tolerant consensus means that every processor that is working correctly eventually decides on the correct output value. If the only way a processor can fail is by omission, and we want to build a system that can tolerate the failure of  $n$  processors, then we need  $n + 1$  processors in the system. Then even if  $n$  processors fail, there will be one correct processor left that can decide the output correctly.

However, if faulty nodes can behave arbitrarily instead of just going offline (i.e., a Byzantine failure) our system requires  $2n + 1$  nodes to tolerate the failure of  $n$  processors, because up to  $n$  faulty processors can decide in favor of the wrong value, so the remaining processors need to know that a majority is deciding in favor of the correct value, so they can all decide identically. In what follows, we will compensate for link failure by equipping each node with a collision detector, and compensate for processor failure by constructing a so-called “replicated state machine.”

## 4 Collision detection

Experimental simulations of wireless networks in which each node is equipped with a collision detector, are promising. Throughput and successful message delivery are almost as good as for idealized “unicast,” in which only one node transmits at a time. (See Figures 2 and 3.)

The principal thesis of the two Chockler, et al., papers is the following:

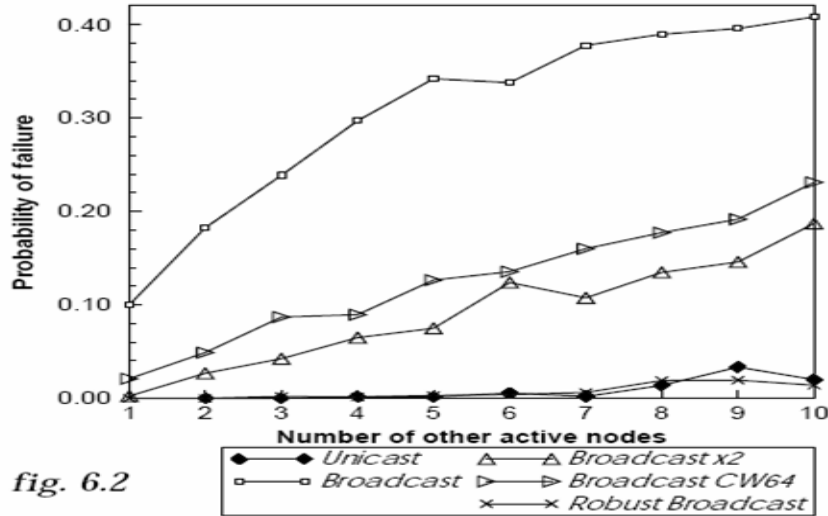


fig. 6.2

Figure 2: Results of simulations of wireless packet traffic. Note that “Robust Broadcast” means broadcast using a MAC protocol that includes collision-detection. Source: “Robust Broadcast: Improving the reliability of broadcast transmissions on CSMA/CA” by J. Tourhilles.

1. Equip each node in the network with a collision detector.
2. Design adaptive algorithms to minimize the broadcast contention.
3. In this way, we can solve the consensus problem for wireless networks, resolving link failure due to collision, and processor failure due to malfunction or battery depletion.

#### 4.1 What do we need in a collision detector?

For a collision detector to be useful, it would need the following two properties.

1. Completeness (all collisions get detected)
2. Accuracy (no false positives; if a collision is reported, it is because that collision actually happened).

However, in a real-world application, perfect completeness and accuracy is too much to hope for. Therefore, we will consider some weakened versions of both. In particular, we will analyze systems with the following properties.

1. Eventual Accuracy (As the algorithm progresses the collision detector will eventually be right—e.g., for every round after a given round  $r_{acc}$ .)
2. Majority-Completeness (A collision is detected if the majority of the messages transmitted in a round is lost.)

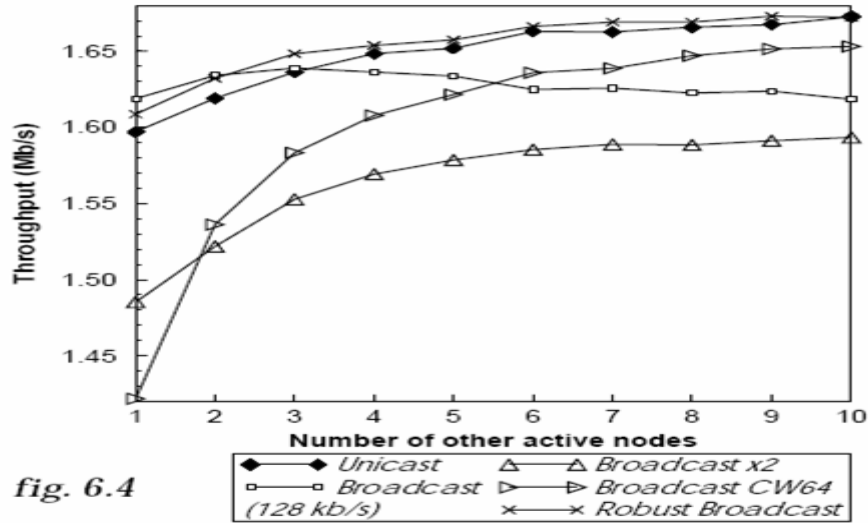


fig. 6.4

Figure 3: Throughput results from the same simulations as Figure 2. Again, note that Robust Broadcast (broadcast with collision detection) performed almost as well as unicast (just one node transmitting at a time).

3. Zero-Completeness (A collision is only detected if *every* message transmitted in a round is lost.)

Let us now look at a specific system model and consensus algorithm that incorporates these definitions.

## 5 The System Model

We will limit discussion to a single-hop wireless network (i.e., all nodes are neighbors of all other nodes). The total number of nodes in the network is unknown, but fixed. (Nodes can fail and leave, but no new node can join.) The initial “input” to the consensus problem will be that each node  $j$  starts with a value  $v_j$  (e.g., either 0 or 1). We will assume at first that the network is entirely synchronous. (The paper later explains how that assumption can be relaxed, but we won’t go into that here, except to explain the different models of synchrony.)

### 5.1 Synchronous vs. asynchronous models

There are three main clock/time models of distributed computing systems:

1. SYNCHRONOUS (All processors have identically-running clocks, and all internode communication takes place at exactly the same time in each round of the algorithm.)
2. ASYNCHRONOUS (Processors have internal clocks, but there is no guaranteed relationship between the clocks. There is no guarantee on communication timing either, except that if a message is sent it will eventually be received, barring link failure.)

3. PARTIALLY SYNCHRONOUS (All processors have clocks, and time matters, but each processor's clock is slightly different, and runs at a slightly different rate. Communication is also partially synchronous.)

The synchronous model is the easiest of the three to write algorithms for, but it is not realistic. It is most useful for proving lower bounds. The asynchronous model also is not realistic for most applications, but it is useful, because any algorithm that executes properly in an asynchronous setting is probably robust. The partially synchronous model is the hardest to deal with mathematically, but is the most realistic.

## 5.2 Algorithm execution

As we assume the system is synchronous, we can divide the execution of the consensus algorithm into synchronous rounds. In each round  $r$ , processor  $p_i$  can do each of the following: (1) broadcast at most one message; (2) receive a subset of the messages transmitted, and (3) perform a state transition.

*Important assumption:* Processors also can *fail* at any point during the execution of the algorithm, except when they are in the middle of broadcasting.

### 5.2.1 Eventual Collision Freedom & Wakeup Service

Collision has been recognized as a problem for wireless communication since the 1970's. At this point, there are several MAC-layer “backoff protocols” to ensure that every wireless node that wants to transmit a message will eventually be able to transmit. The Chockler, et al., model captures that mathematically by assuming that even if a collision happens at time  $t$ , if we wait long enough, collision-free communication can once again take place.

Formally, there exists a positive integer  $b$ , such that in each execution, there exists a round  $r_{ecf}$  so that the following is satisfied: for each round  $r \geq r_{ecf}$ , if at most  $b$  nodes broadcast messages in  $r$ , then all correct nodes receive all the messages that have been broadcast in  $r$ .

To take advantage of Eventual Collision Freedom, the algorithm uses a “Wakeup Service,” which determines which nodes should transmit in the current round of execution. (This idea is similar to the “query set” concept we saw at the beginning of the semester in the Komlós-Greenberg paper.)

Formally, there exists a round  $r_{wake}$  such that for each  $r \geq r_{wake}$ , the wake-up service provides good advice for round  $r$  (meaning it allows all waiting nodes to transmit without causing further collisions).

### 5.2.2 The consensus algorithm

A high-level description of the consensus algorithm would be: Each node starts with an arbitrary input. Each such input value is a member of a fixed set  $V$ . At the end of the algorithm's execution, each node outputs a value. This process must satisfy the following three conditions.

1. AGREEMENT (All correctly-operating nodes decide on the same value.)
2. VALIDITY (If a node decides on a value  $v$ , then  $v$  must have been the initial assignment to one of the nodes in the network.)
3. TERMINATION (All correct nodes eventually decide.)

## 6 The main algorithm, and lower-bound results

There are two phases to the main consensus algorithm, as follows.

### 1. PROPOSAL PHASE

- (a) Every node sends out its estimate of the correct value.
- (b) The passive nodes do not broadcast.
- (c) If a node hears no collisions, it updates its estimate to the minimum value received.
- (d) If a node hears a collision, or hears more than one estimate, it enters state VETO.

### 2. VETO PHASE

- (a) Every node in state VETO broadcasts a veto message.
- (b) If a node hears no veto messages and detects no collisions, then the node can decide.

The basic idea is that each node has two opportunities to veto a decision, and correct nodes will not decide if they hear a veto or detect a collision. This guarantees that nodes will not decide until they heard from all other nodes, so all correct nodes will decide on the same value.

### 6.1 Lower bound results

Here is a summary of the paper's lower bound results, for different strengths of accuracy and completeness, relative to the possibility of eventual collision freedom.

Accuracy/completeness	Eventual Collision Freedom	No Collision Freedom
Accurate/Complete	$\Theta(1)$	$\Theta(\log  V )$
Acc/Maj-Complete	$\Theta(1)$	$\Theta(\log  V )$
Acc/Zero-Complete	$\Theta(\log  V )$	$\Theta(\log  V )$
Event-Acc/Complete	$\Theta(1)$	Impossible
Event-Acc/Maj-Complete	$\Theta(1)$	Impossible
Event-Acc/Zero-Complete	$\Theta(\log  V )$	Impossible

I found it very interesting that the parameter  $n$  (the number of nodes in the network graph) appears nowhere in these lower bounds. If the consensus problem is solvable at all, it is solvable in constant time, or in time proportional to the size of  $V$ . The total number of nodes in the network is not a significant factor.

Because of Eventual Collision Freedom and the Wake-Up Service, the algorithm is guaranteed transmission will be possible after waiting linear-many rounds. Then it's like a binary search: there are at most  $|V|$ -many different groups of disagreeing nodes. In each round after  $r_{ecf}$ , nodes take on the minimum value they hear transmitted, eliminating up to  $|V|/2$  groups of disagreeing nodes. So it only takes  $\mathcal{O}(\log |V|)$  rounds to achieve consensus.

## 6.2 Weak-validity consensus

By weakening the statement of the consensus problem, we can overcome some of the lower-bound results. For example, we could include a “default value” in the consensus process, and say that a node could output the default value, instead of having to output the same value as all other correct nodes. Example: a default value that means “abort transaction” in the event stronger consensus is not achievable. One application for this would be in a large relational database, where multiple software processes must agree that their step in some function was completed before the database could commit to erasing old information in favor of the new information.

We must be careful to maintain nontriviality here, because if all processors simply choose the default value all the time, we're not really solving anything. One way to avoid triviality would be to require that a processor can only choose the default value if it detects a collision.

There is an algorithm that assumes accurate and complete collision detection and achieves weak validity consensus in two rounds.

## 7 Performance Evaluation

While the paper provides no proof of efficiency for multi-hop networks, simulation results demonstrated the same consensus algorithm scaled well from a single-hop to multi-hop setting. See Figures 4 and 5.

## 8 Replicated State Machines for Wireless Networks

A *replicated state machine* (RSM) is a general method for implementing a fault-tolerant service by replicating identical states on multiple processors, and coordinating client interactions with those replicas. The processors together behave as an *atomic object*.

An atomic object behaves like a single computational device, when viewed from the exterior. On the “inside,” it may be a collection of independent processes or nodes that coordinate to make a single decision on what actions they should take.

Most research into replicated state machines has been on distributed systems with a central brain and a wired backbone. The second paper we are considering today introduces a new line of research: RSM algorithms that try to be robust in the face of the special problems of wireless ad hoc networks. Previous RSM models have either been unsuitable for anonymous

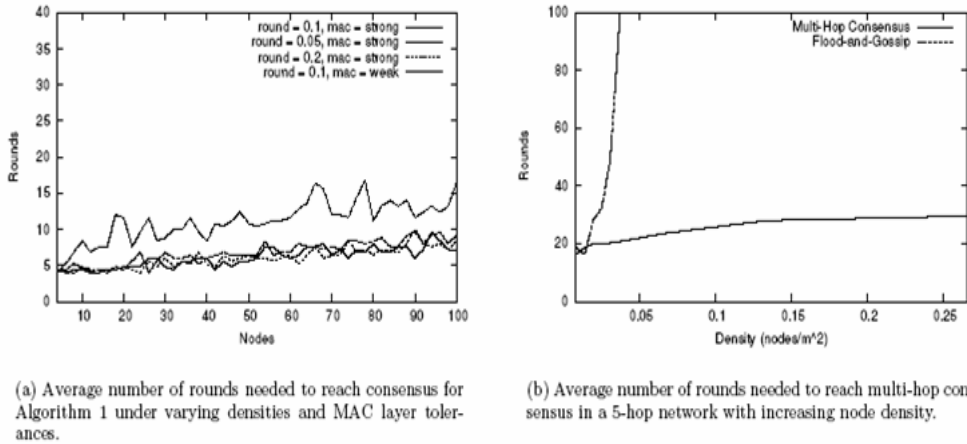


Figure 4: Simulations. Each data point is the average of five independent simulation runs.

settings (they need to know how many participants are in the network, they need each node to have a unique identifier, they cannot handle dynamic joins and leaves), or assumed reliable, collision-free communication.

Much as adding collision-detection allowed for an appropriate solution to the consensus problem, we can construct an RSM with collision-detection suitable to the setting of wireless ad hoc networks, i.e., anonymous nodes, unexpected joins and leaves, or temporary link failure due to broadcast collision.

Broadly, we can say that an RSM algorithm should accomplish at least the following objectives.

1. It is collision-tolerant (safe at all times).
2. It adapts to changing numbers of nodes.
3. It is efficient (all protocol message are of constant size, and the execution of one state machine round takes only constant steps).
4. It works with ad hoc deployments.
5. It tolerates an arbitrary number of crashes.

### 8.1 Collision-Aware RSM's

A collision-aware RSM is a tuple of form  $\langle V, v_0, P, outputs, \delta \rangle$ , in which the parameters have the following definitions.

- $V$  the set of legal states
- $v_0$  the initial state of the RSM
- $P$  the set of "proposals" or inputs
- $outputs$  a set of values to learn
- $\delta$  a transition function of form  
 $\delta : V \times P \cup \{\text{collision}\} \rightarrow V \times outputs$

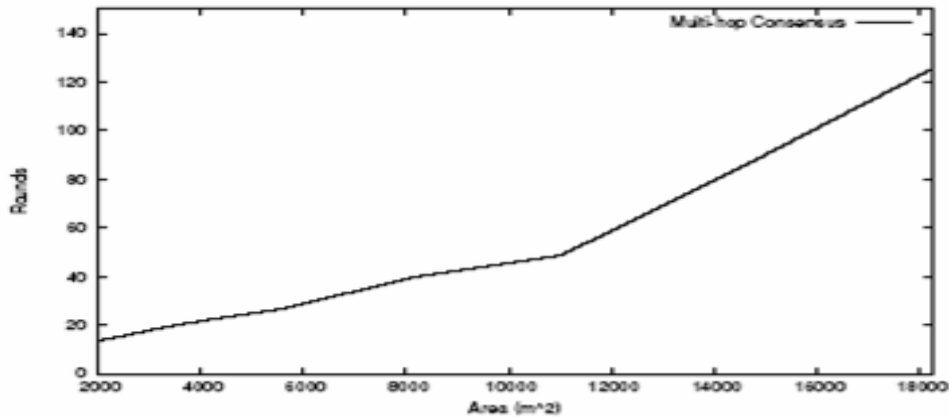


Figure 2: Average number of rounds needed to reach multi-hop consensus for a density of  $0.02667 \text{ nodes/m}^2$  (approx. 6 nodes per single-hop area) and increasing network area.

Figure 5: Figure 2 from the original paper.

We say that algorithm  $A$  implements a collision-aware RSM if the resulting state-machine rounds are consistent with a valid execution of the automaton. Formally: the sequence of states and the learned values are all consistent with the transition function, and the algorithm detects collisions, respects that communication does not take place freely in the event of collisions, and does not rely on work done by processors that have failed.

The system model in the second paper makes several assumptions that are identical to the first paper. One of these is the existence of Eventual Collision Freedom. So, as before, we require that after round  $r_{ecf}$  and round  $r_{acc}$ , the network experiences no further collisions, and all waiting nodes will eventually be able to transmit.

## 9 RSM protocols for fault-tolerant consensus

This RSM model assigns nodes one of three different roles: proposers, learners and replicas. The proposers suggest values for final consensus decision; the learners learn the values and are responsible for deciding correctly; and the replicas store information redundantly, so that consensus can be solved correctly in the event of faults. The relationship between these three roles is depicted in Figure 6.

Each round of algorithm execution contains three protocols, one for each type of RSM node. They occur in the following order.

### 1. PROPOSER PROTOCOL

- (a) During the first phase of the state-machine round, the proposers broadcast their proposals.

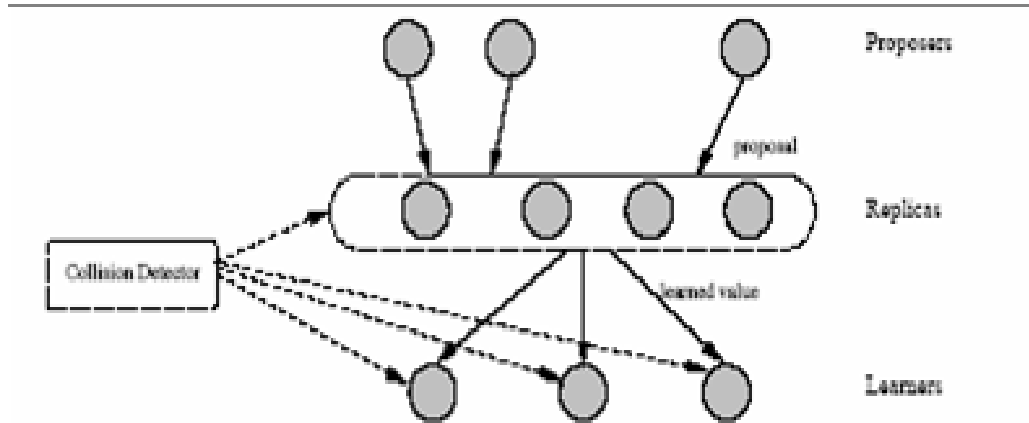


Figure 6: An RSM for wireless networks. Nodes can have one of three different jobs in a given execution: proposers, replicas or listeners.

## 2. REPLICAS PROTOCOL

- (a) *Ballot phase:* All active replicas broadcast a ballot, i.e., what they think the correct value should be.
- (b) *Veto-One phase:* If a replica detects a collision in the ballot phase, it broadcasts a veto message now.
- (c) *Veto-Two phase:* If a replica detects a collision or a veto in the Veto-One phase, it broadcasts a veto message now.

## 3. LEARNER PROTOCOL

- (a) Listen to the messages sent by the replicas. If a learner detects a collision, it outputs  $\{\text{collision}\}$ .

There are two veto rounds to improve the efficiency of the algorithm. With only one veto round, each processor would need to keep in memory a history of all previous consensus data. This way, each state-machine round, the history can be wiped clean, because each new instance of “consensus” effectively supersedes all previous instances.

In order for all learners to learn the correct value, they need to be certain that they heard all possible transmissions from the replicas. If a collision occurs, it is possible that the learners heard some messages but not others. Therefore, the algorithm uses a four-color code (green, yellow, orange and red) to classify heard values. Values heard at a given round are colored according to the following table.

ballot	veto-1	veto-2	Replica	Learner
✓	✓	✓	Green	Green
✓	✓	X	Yellow	Red
✓	X	X	Orange	Red
X	X	X	Red	Red

A checkmark means that no collision was detected during that phase of the round, while an X means that at least one collision was detected during that phase. All values heard by either replica or learner are colored with the appropriate color in the table.

## 10 Sketch of formal RSM algorithm

To use the round/value colors effectively, each node is equipped with a table of pointers, so it can reconstruct the last “clear” round, i.e., the last time it heard values and detected no collisions. A graphic example of this process appears in Figure 7.

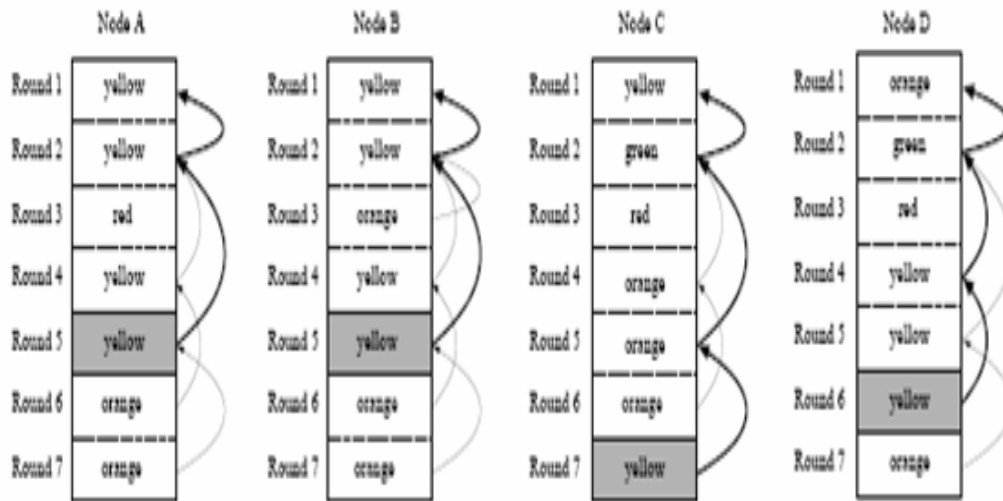


Figure 7: A sample RSM execution.

A high-level overview of the complete RSM algorithm is as follows.

1. PROPOSE: The proposers send out proposals. The replicas receive the proposals and assemble a ballot.
2. 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 RSM round red.
3. VETO-ONE: Replicas broadcast vetoes if they did not receive a ballot. If a replica or learner then detects or collision or receives a veto, they color the round orange. Otherwise, if the round is not red or orange, the replicas accept the round and update their pointers.
4. VETO-TWO: 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 RSM round yellow. If the round is still green, the replicas and learners commit to accepting this RSM round. Learners output appropriately. Otherwise, they simply output {collision}.

The main function of the algorithm is to ensure that a node decides only if it is certain that it has heard all transmission without collision. Because of the assumption of eventual collision freedom and eventual accuracy, every correct node will eventually decide correctly. The existence of replicas ensures that the network can solve the consensus problem even when saddled with many faulty processors.

## 11 Conclusion

One question that comes to mind is whether these results are just theoretical, or whether they can be implemented. In particular: do any real collision detectors exist?

I could find no clearly effective collision-detection hardware after lengthy search. Perhaps more striking, all the references in the Chockler et al papers either simulate collision-detection, or make reference exclusively to software-process collision detection, such as RTS and CTS messages in MACAW. Therefore, this work is a guide for the future, an example of one way to overcome the challenges of collision in ad hoc wireless networks.

In sum, consensus algorithms and Replicated State Machines have been critical tools in distributed computing since the 1970s. The special conditions of wireless ad hoc networks prevent the direct translation of the same computational models. The theoretical results of equipping nodes with collision-detection is a promising direction for future research.