Fault-Tolerant Consensus with an Abstract MAC Layer

Calvin Newport[1]
Georgetown University, Washington, D.C., USA
cnewport@cs.georgetown.edu

Peter Robinson[2]
McMaster University, Hamilton, Ontario, Canada
peter.robinson@mcmaster.ca

Abstract

In this paper, we study fault-tolerant distributed consensus in wireless systems. In more detail, we produce two new randomized algorithms that solve this problem in the abstract MAC layer model, which captures the basic interface and communication guarantees provided by most wireless MAC layers. Our algorithms work for any number of failures, require no advance knowledge of the network participants or network size, and guarantee termination with high probability after a number of broadcasts that are polynomial in the network size. Our first algorithm satisfies the standard agreement property, while our second trades a faster termination guarantee in exchange for a looser agreement property in which most nodes agree on the same value. These are the first known fault-tolerant consensus algorithms for this model. In addition to our main upper bound results, we explore the gap between the abstract MAC layer and the standard asynchronous message passing model by proving fault-tolerant consensus is impossible in the latter in the absence of information regarding the network participants, even if we assume no faults, allow randomized solutions, and provide the algorithm a constant-factor approximation of the network size.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases abstract MAC layer, wireless networks, consensus, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.38

Related Version A full version of the paper is available at [42], `https://www.cas.mcmaster.ca/robinson/random-aml.pdf`.

1 Introduction

Consensus provides a fundamental building block for developing reliable distributed systems [23–25]. Accordingly, it is well studied in many different system models [36]. Until recently, however, little was known about solving this problem in distributed systems made up of devices communicating using commodity wireless cards. Motivated by this knowledge gap, this paper studies consensus in the *abstract MAC layer* model, which abstracts the basic behavior and guarantees of standard wireless MAC layers. In recent work [41], we proved deterministic fault-tolerant consensus is impossible in this setting. In this paper, we describe and analyze the first known randomized fault-tolerant consensus algorithms for this well-motivated model.

[1] Calvin Newport acknowledges the support of the National Science Foundation, award number 1733842.
[2] Peter Robinson acknowledges the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2018-06322.

The Abstract MAC Layer. Most existing work on distributed algorithms for wireless networks assumes low-level synchronous models that force algorithms to directly grapple with issues caused by contention and signal fading. Some of these models describe the network topology with a graph (c.f., [8, 16, 20, 28, 32, 38]), while others use signal strength calculations to determine message behavior (c.f., [17, 21, 26, 27, 37, 39]).

As also emphasized in [41], these models are useful for asking foundational questions about distributed computation on shared channels, but are not so useful for developing algorithmic strategies suitable for deployment. In real systems, algorithms typically do not operate in synchronous rounds and they are not provided unmediated access to the radio. They must instead operate on top of a general-purpose MAC layer which is responsible for many network functions, including contention management, rate control, and co-existence with other network traffic.

Motivated by this reality, in this paper we adopt the *abstract MAC layer* model [34], an asynchronous broadcast-based communication model that captures the basic interfaces and guarantees provided by common existing wireless MAC layers. In more detail, if you provide the abstract MAC layer a message to broadcast, it will eventually be delivered to nearby nodes in the network. The specific means by which contention is managed – e.g., CSMA, TDMA, uniform probabilistic routines such as DECAY [8] – is abstracted away by the model. At some point after the contention management completes, the abstract MAC layer passes back an *acknowledgment* indicating that it is ready for the next message. This acknowledgment contains no information about the number or identities of the message recipient.

(In the case of the MAC layer using CSMA, for example, the acknowledgment would be generated after the MAC layer detects a clear channel. In the case of TDMA, the acknowledgment would be generated after the device's turn in the TDMA schedule. In the case of a probabilistic routine such as DECAY, the acknowledgment would be generated after a sufficient number of attempts to guarantee successful delivery to all receivers with high probability.)

The abstract MAC abstraction, of course, does not attempt to provide a detailed representation of any specific existing MAC layer. Real MAC layers offer many more modes and features then is captured by this model. In addition, the variation studied in this paper assumes messages are always delivered, whereas more realistic variations would allow for occasional losses.

This abstraction, however, still serves to capture the fundamental dynamics of real wireless application design in which the lower layers dealing directly with the radio channel are separated from the higher layers executing the application in question. An important goal in studying this abstract MAC layer, therefore, is attempting to uncover principles and strategies that can close the gap between theory and practice in the design of distributed systems deployed on standard layered wireless architectures.

Our Results. In this paper, we studied randomized fault-tolerant consensus algorithms in the abstract MAC layer model. In more detail, we study binary consensus and assume a single-hop network topology. Notice, our use of randomization is necessary, as deterministic consensus is impossible in the abstract MAC layer model in the presence of even a single fault (see our generalization of FLP from [41]).

To contextualize our results, we note that the abstract MAC layer model differs from standard asynchronous message passing models in two main ways: (1) the abstract MAC layer model provides the algorithm no advance information about the network size or membership,

requiring nodes to communicate with a blind broadcast primitive instead of using point-to-point channels, (2) the abstract MAC layer model provides an acknowledgment to the broadcaster at some point after its message has been delivered to all of its neighbors. This acknowledgment, however, contains no information about the number or identity of these neighbors (see above for more discussion of this fundamental feature of standard wireless MAC layers).

Most randomized fault-tolerant consensus algorithms in the asynchronous message passing model strongly leverage knowledge of the network. A strategy common to many of these algorithms, for example, is to repeatedly collect messages from at least $n - f$ nodes in a network of size n with at most f crash failures (e.g., [9]). This strategy does not work in the abstract MAC layer model as nodes do not know n.

To overcome this issue, we adapt an idea introduced in early work on fault-tolerant consensus in the asynchronous shared memory model: *counter racing* (e.g., [5, 12]). At a high-level, this strategy has nodes with initial value 0 advance a shared memory counter associated with 0, while nodes with initial value 1 advance a counter associated with 1. If a node sees one counter get ahead of the other, they adopt the initial value associated with the larger counter, and if a counter gets sufficiently far ahead, then nodes can decide.

Our first algorithm (presented in Section 3) implements a counter race of sorts using the acknowledged blind broadcast primitive provided by the model. Roughly speaking, nodes continually broadcast their current proposal and counter, and update both based on the pairs received from other nodes. Proving safety for this type of strategy in shared memory models is simplified by the atomic nature of register accesses. In the abstract MAC layer model, by contrast, a broadcast message is delivered non-atomically to its recipients, and in the case of a crash, may not arrive at some recipients at all.[3] Our safety analysis, therefore, requires novel analytical tools that tame a more diverse set of possible system configurations.

To achieve liveness, we use a technique loosely inspired by the randomized delay strategy introduced by Chandra in the shared memory model [12] . In more detail, nodes probabilistically decide to replace certain sequences of their counter updates with *nop* placeholders. We show that if these probabilities are adapted appropriately, the system eventually arrives at a state where it becomes likely for only a single node to be broadcasting updates, allowing progress toward termination.

Formally, we prove that with high probability in the network size n, the algorithm terminates after $O(n^3 \log n)$ broadcasts are scheduled. This holds regardless of which broadcasts are scheduled (i.e., we do not impose a fairness condition), and regardless of the number of faults. The algorithm, as described, assumes nodes are provided unique IDs that we treat as comparable black boxes (to prevent them from leaking network size information). We subsequently show how to remove that assumption by describing an algorithm that generates unique IDs in this setting with high probability.

Our second algorithm (presented in Section 4) trades a looser agreement guarantee for more efficiency. In more detail, we describe and analyze a solution to *almost-everywhere* agreement [18], that guarantees most nodes agree on the same value. This algorithm terminates after $O(n^2 \log^4 n \log \log n)$ broadcasts, which is a linear factor faster than our first algorithm (ignoring log factors). The almost-everywhere consensus algorithm consists of two phases. The first phase is used to ensure that almost all nodes obtain a good approximation

[3] We note that register simulations are also not an option in our model for two reasons: standard simulation algorithms require knowledge of n and a majority correct nodes, whereas we assume no knowledge of n and wait-freedom.

of the network size. In the second phase, nodes use this estimate to perform a sequence of broadcasts meant to help spread their proposal to the network. Nodes that did not obtain a good estimate in Phase 1 will leave Phase 2 early. The remaining nodes, however, can leverage their accurate network size estimates to probabilistically sample a subset to actively participate in each round of broadcasts. To break ties between simultaneously active nodes, each chooses a random rank using the estimate obtained in Phase 1. We show that with high probability, after not too long, there exists a round of broadcasts in which the first node receiving its acknowledgment is both active and has the minimum rank among other active nodes – allowing its proposal to spread to all remaining nodes.

Finally, we explore the gap between the abstract MAC layer model and the related asynchronous message passage passing model. We prove (in Section 5) that fault-tolerant consensus is impossible in the asynchronous message passing model in the absence of knowledge of network participants, even if we assume no faults, allow randomized algorithms, and provide a constant-factor approximation of n. This differs from the abstract MAC layer model where we solve this problem without network participant or network size information, and assuming crash failures. This result implies that the fact that broadcasts are acknowledged in the abstract MAC layer model is crucial to overcoming the difficulties induced by limited network information.

Related Work. Consensus provides a fundamental building block for reliable distributed computing [23–25]. It is particularly well-studied in asynchronous models [2, 35, 40, 44].

The abstract MAC layer approach[4] to modeling wireless networks was introduced in [33] (later expanded to a journal version [34]), and has been subsequently used to study several different problems [14, 15, 29, 30, 41]. The most relevant of this related work is [41], which was the first paper to study consensus in the abstract MAC layer model. This previous paper generalized the seminal FLP [19] result to prove deterministic consensus is impossible in this model even in the presence of a single failure. It then goes on to study deterministic consensus in the absence of failures, identifying the pursuit of fault-tolerant *randomized* solutions as important future work – the challenge taken up here.

We note that other researchers have also studied consensus using high-level wireless network abstractions. Vollset and Ezhilchelvan [45], and Alekeish and Ezhilchelvan [4], study consensus in a variant of the asynchronous message passing model where pairwise channels come and go dynamically – capturing some behavior of mobile wireless networks. Their correctness results depend on detailed liveness guarantees that bound the allowable channel changes. Wu et al. [46] use the standard asynchronous message passing model (with unreliable failure detectors [13]) as a stand-in for a wireless network, focusing on how to reduce message complexity (an important metric in a resource-bounded wireless setting) in solving consensus.

A key difficulty for solving consensus in the abstract MAC layer model is the absence of advance information about network participants or size. These constraints have also been studied in other models. Ruppert [43], and Bonnet and Raynal [10], for example, study the amount of extra power needed (in terms of shared objects and failure detection, respectively) to solve wait-free consensus in *anonymous* versions of the standard models. Attiya et al. [6] describe consensus solutions for shared memory systems without failures or unique ids. A

[4] There is no *one* abstract MAC layer model. Different studies use different variations. They all share, however, the same general commitment to capturing the types of interfaces and communication/timing guarantees that are provided by standard wireless MAC layers

series of papers [3, 11, 22], starting with the work of Cavin et al. [11], study the related problem of *consensus with unknown participants* (CUPs), where nodes are only allowed to communicate with other nodes whose identities have been provided by a *participant detector* formalism.

Closer to our own model is the work of Abboud et al. [1], which also studies a single hop network where nodes broadcast messages to an unknown group of network participants. They prove deterministic consensus is impossible in these networks under these assumptions without knowledge of network size. In this paper, we extend these existing results by proving this impossibility still holds even if we assume randomized algorithms and provided the algorithm a constant-factor approximation of the network size. This bound opens a sizable gap with our abstract MAC layer model in which consensus is solvable without this network information.

We also consider almost-everywhere (a.e.) agreement [18], a weaker variant of consensus, where a small number of nodes are allowed to decide on conflicting values, as long as a sufficiently large majority agrees. Recently, a.e. agreement has been studied in the context of peer-to-peer networks (c.f. [7, 31]), where the adversary can isolate small parts of the network thus rendering (everywhere) consensus impossible. We are not aware of any prior work on a.e. agreement in the wireless settings.

2 Model and Problem

In this paper, we study a variation of the *abstract MAC layer* model, which describes system consisting of a single hop network of $n \geq 1$ computational devices (called *nodes* in the following) that communicate wirelessly using communication interfaces and guarantees inspired by commodity wireless MAC layers.

In this model, nodes communicate with a *bcast* primitive that guarantees to eventually deliver the broadcast message to all the other nodes (i.e., the network is single hop). At some point after a given *bcast* has succeeded in delivering a message to all other nodes, the broadcaster receives an *ack* informing it that the broadcast is complete (as detailed in the introduction, this captures the reality that most wireless contention management schemes have a definitive point at which they know a message broadcast is complete). This acknowledgment contains no information about the number or identity of the receivers.

We assume a node can only broadcast one message at a time. That is, once it invokes *bcast*, it cannot broadcast another message until receiving the corresponding *ack* (formally, overlapping messages are discarded by the MAC layer). We also assume any number of nodes can permanently stop executing due to crash failures. As in the classical message passing models, a crash can occur during a broadcast, meaning that some nodes might receive the message while others do not.

This model is event-driven with the relevant events scheduled asynchronously by an arbitrary *scheduler*. In more detail, for each node u, there are four event types relevant to u that can be scheduled: $init_u$ (which occurs at the beginning of an execution and allows u to initialize), $recv(m)_u$ (which indicates that u has received message m broadcast from another node), $ack(m)_u$ (which indicates that the message m broadcast by u has been successfully delivered), and $crash_u$ (which indicates that u is crashed for the remainder of the execution).

A distributed algorithm specifies for each node u a finite collection of steps to execute for each of the non-*crash* event types. When one of these events is scheduled by the scheduler, we assume the corresponding steps are executed atomically at the point that the event is

scheduled. Notice that one of the steps that a node u can take in response to these events is to invoke a $bcast(m)_u$ primitive for some message m. When an event includes a *bcast* primitive we say it is *combined* with a broadcast.[5]

We place the following constraints on the scheduler. It must start each execution by scheduling an *init* event for each node; i.e., we study the setting where all participating nodes are activated at the beginning of the execution. If a node u invokes a valid $bcast(m)_u$ primitive, then for each $v \neq u$ that is not crashed when the broadcast primitive is invoked, the scheduler must subsequently either schedule a single $recv(m)_v$ or $crash_v$ event at v. At some point after these events are scheduled, it must then eventually schedule an $ack(m)_u$ event at u. These are the only *recv* and *ack* events it schedules (i.e., it cannot create new messages from scratch or cause messages to be received/acknowledged multiple times). If the scheduler schedules a $crash_v$ event, it cannot subsequently schedule any future events for u.

We assume that in making each event scheduling decision, the scheduler can use the schedule history as well as the algorithm definition, but it does not know the nodes' private states (which includes the nodes' random bits). When the scheduler schedules an event that triggers a broadcast (making it a combined event), it is provided this information so that it knows it must now schedule receive events for the message. We assume, however, that the scheduler does not learn the *contents* of the broadcast message.[6]

Given an execution α, we say the *message schedule* for α, also indicated $msg[\alpha]$, is the sequence of message events (i.e., *recv*, *ack*, and *crash*) scheduled in the execution. We assume that a message schedule includes indications of which events are combined with broadcasts.

The Consensus Problem. In this paper, we study binary consensus with probabilistic termination. In more detail, at the beginning of an execution each node is provided an *initial value* from $\{0, 1\}$ as input. Each node has the ability to perform a single irrevocable *decide* action for either value 0 or 1. To solve consensus, an algorithm must guarantee the following three properties: (1) *agreement*: no two nodes decide different values; (2) *validity*: if a node decides value b, then at least one node started with initial value b; and (3) *termination (probabilistic)*: every non-crashed node decides with probability 1 in the limit.

Studying finite termination bounds is complicated in asynchronous models because the scheduler can delay specific nodes taking steps for arbitrarily long times. In this paper, we circumvent this issue by proving bounds on the number of scheduled events before the system reaches a *termination state* in which every non-crashed node has: (a) decided; or (b) will decide whenever the scheduler gets around to scheduling its next *ack* event.

Finally, in addition to studying consensus with standard agreement, we also study *almost-everywhere* agreement, in which only a specified majority fraction (typically a $1-o(n)$ fraction of the n total nodes) must agree.

---

[5] Notice, we can assume without loss of generality, that the steps executed in response to an event never invoke more than a single *bcast* primitive, as any additional broadcasts invoked at the same time would lead to the messages being discarded due to the model constraint that a node must receive an *ack* for the current message before broadcasting a new message.

[6] This adversary model is sometimes called *message oblivious* and it is commonly considered a good fit for schedulers that control network behavior. This follows because it allows the scheduler to adapt the schedule based on the number of messages being sent and their sources – enabling it to model contention and load factors. One the other hand, there is not good justification for the idea that this schedule should somehow also depend on the specific bits contained in the messages sent. Notice, our liveness proof specifically leverages the message oblivious assumption as it prevents the scheduler from knowing which nodes are sending updates and which are sending *nop* messages.

Algorithm 1 Counter Race Consensus (for node u with UID id_u and initial value v_u)

Initialization:
$c_u \leftarrow 0$
$n_u \leftarrow 2$
$C_u \leftarrow \{(id_u, c_u, v_u)\}$
$peers \leftarrow \{id_u\}$
$phase \leftarrow 0$
$active \leftarrow true$
$decide \leftarrow -1$
$k \leftarrow 3$
$c \leftarrow k + 3$
bcast(nop, id_u, n_u)

On Receiving $ack(m)$:
$phase \leftarrow phase + 1$
if $m = (decide, b)$ **then**
  decide(b) and **halt**()
else
  $newm \leftarrow \perp$
  $C'_u \leftarrow C_u$
  $\hat{c}_u^{(0)} \leftarrow$ max counter in C'_u paired with value 0 (default to 0 if no such elements)
  $\hat{c}_u^{(1)} \leftarrow$ max counter in C'_u paired with value 1 (default to 0 if no such elements)
  if $\hat{c}_u^{(0)} > \hat{c}_u^{(1)}$ **then** $v_u \leftarrow 0$
  else if $\hat{c}_u^{(1)} > \hat{c}_u^{(0)}$ **then** $v_u \leftarrow 1$
  if $\hat{c}_u^{(0)} \geq \hat{c}_u^{(1)} + k$ **or** $decide = 0$ **then** $newm \leftarrow (decide, 0)$
  else if $\hat{c}_u^{(1)} \geq \hat{c}_u^{(0)} + k$ **or** $decide = 1$ **then** $newm \leftarrow (decide, 1)$
  if $newm = \perp$ **then**
    if $\max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\} \leq c_u$ **and** $m \neq nop$ **then** $c_u \leftarrow c_u + 1$
    else if $\max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\} > c_u$ **then** $c_u \leftarrow max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\}$
    update $(id_u, *, *)$ element in C_u with new c_u and v_u
    $newm \leftarrow (counter, id_u, c_u, v_u, n_u)$
  if $phase \,\%\, c = 1$ **then** with probability $1/n_u$ $active \leftarrow true$ otherwise $active \leftarrow false$
  if $newm = (decide, *)$ **or** $active = true$ **then**
    bcast$(newm)$
  else
    bcast(nop, id_u, n_u)

On Receiving Message m:
updateEstimate(m)
if $m = (decide, b)$ **then**
  $decide \leftarrow b$
else if $m = (counter, id, c, v, n')$ **then**
  if $\exists c', v'$ such that $(id, c', v') \in C_u$ **then**
    remove (id, c', v') from C_u
  add (id, c, v) to C_u

Algorithm 2 The **updateEstimate**(m) subroutine called by Counter Race Consensus during $recv(m)$ event.

if m contains a UID id and network size estimate n' **then**
  $peers \leftarrow peers \cup \{id\}$
  $n_u \leftarrow \max\{n_u, |peers|, n'\}$

3 Upper Bound

Here we describe analyze our first randomized binary consensus algorithm: *counter race consensus* (see Algorithms 1 and 2 for pseudocode, and Section 3.1 for a high-level description of its behavior). This algorithm assumes no advance knowledge of the network participants or network size. Nodes are provided unique IDs, but these are treated as comparable black boxes, preventing them from leaking information about the network size. (We will later discuss how to remove the unique ID assumption.) It tolerates any number of crash faults. The detailed proofs can be found in the full paper [42].

3.1 Algorithm Description

The counter race consensus algorithm is described in pseudocode in the figures labeled Algorithm 1 and 2. Here we summarize the behavior formalized by this pseudocode.

The core idea of this algorithm is that each node u maintains a counter c_u (initialized to 0) and a proposal v_u (initialized to its consensus initial value). Node u repeatedly broadcasts c_u and v_u, updating these values before each broadcast. That is, during the *ack* event for its last broadcast of c_u and v_u, node u will apply a set of *update rules* to these values. It then concludes the *ack* event by broadcasting these updated values. This pattern repeats until u arrives at a state where it can safely commit to deciding a value.

The update rules and decision criteria applied during the *ack* event are straightforward. Each node u first calculates $\hat{c}_u^{(0)}$, the largest counter value it has sent or received in a message containing proposal value 0, and $\hat{c}_u^{(1)}$, the largest counter value it has sent or received in a message containing proposal value 1.

If $\hat{c}_u^{(0)} > \hat{c}_u^{(1)}$, then u sets $v_u \leftarrow 0$, and if $\hat{c}_u^{(1)} > \hat{c}_u^{(0)}$, then u sets $v_u \leftarrow 1$. That is, u adopts the proposal that is currently "winning" the counter race (in case of a tie, it does not change its proposal).

Node u then checks to see if either value is winning by a large enough margin to support a decision. In more detail, if $\hat{c}_u^{(0)} \geq \hat{c}_u^{(1)} + 3$, then u commits to deciding 0, and if $\hat{c}_u^{(1)} \geq \hat{c}_u^{(0)} + 3$, then u commits to deciding 1.

What happens next depends on whether or not u committed to a decision. If u did *not* commit to a decision (captured in the **if** $newm = \perp$ **then** conditional), then it must update its counter value. To do so, it compares its current counter c_u to $\hat{c}_u^{(0)}$ and $\hat{c}_u^{(1)}$. If c_u is smaller than one of these counters, it sets $c_u \leftarrow \max\{\hat{c}_u^{(0)}, \hat{c}_u^{(1)}\}$. Otherwise, if c_u is the largest counter that u has sent or received so far, it will set $c_u \leftarrow c_u + 1$. Either way, its counter increases. At this point, u can complete the *ack* event by broadcasting a message containing its newly updated c_u and v_u values.

On the other hand, if u committed to deciding value b, then it will send a $(decide, b)$ message to inform the other nodes of its decision. On subsequently receiving an *ack* for this message, u will decide b and halt. Similarly, if u ever receives a $(decide, b)$ message from *another* node, it will commit to deciding b. During its next *ack* event, it will send its

own $(decide, b)$ message and decide and halt on its corresponding *ack*. That is, node u will not decide a value until it has broadcast its commitment to do so, and received an *ack* on the broadcast.

The behavior described above guarantees agreement and validity. It is not sufficient, however, to achieve liveness, as an ill-tempered scheduler can conspire to keep the race between 0 and 1 too close for a decision commitment. To overcome this issue we introduce a random delay strategy that has nodes randomly step away from the race for a while by replacing their broadcast values with *nop* placeholders ignored by those who receive them. Because our adversary does not learn the *content* of broadcast messages, it does not know which nodes are actively participating and which nodes are taking a break (as in both cases, nodes continually broadcast messages) – thwarting its ability to effectively manipulate the race.

In more detail, each node u partitions its broadcasts into *groups* of size 6. At the beginning of each such group, u flips a weighted coin to determine whether or not to replace the counter and proposal values it broadcasts during this group with *nop* placeholders – eliminating its ability to affect other nodes' counter/proposal values. As we will later elaborate in the liveness analysis, the goal is to identify a point in the execution in which a single node v is broadcasting its values while all other nodes are broadcasting *nop* values – allowing v to advance its proposal sufficiently far ahead to win the race.

To be more specific about the probabilities used in this logic, node u maintains an estimate n_u of the number of nodes in the network. It replaces values with *nop* placeholders in a given group with probability $1/n_u$. (In the pseudocode, the *active* flag indicates whether or not u is using *nop* placeholders in the current group.) Node u initializes n_u to 2. It then updates it by calling the *updateEstimate* routine (described in Algorithm 2) for each message it receives.

There are two ways for this routine to update n_u. The first is if the number of unique IDs that u has received so far (stored in $peers$) is larger than n_u. In this case, it sets $n_u \leftarrow |peers|$. The second way is if it learns another node has an estimate $n' > n_u$. In this case, it sets $n_u \leftarrow n'$. Node u learns about other nodes' estimates, as the algorithm has each node append its current estimate to all of its messages (with the exception of *decide* messages). In essence, the nodes are running a network size estimation routine parallel to its main counter race logic – as nodes refine their estimates, their probability of taking useful breaks improves.

3.2 Safety

We begin our analysis by proving that our algorithm satisfies the agreement and validity properties of the consensus problem. Validity follows directly from the algorithm description. Our strategy to prove agreement is to show that if any node sees a value b with a counter at least 3 ahead of value $1 - b$ (causing it to commit to deciding b), then b is the only possible decision value. Race arguments of this type are easier to prove in a shared memory setting where nodes work with objects like atomic registers that guarantee linearization points. In our message passing setting, by contrast, in which broadcast messages arrive at different receivers at different times, we will require more involved definitions and operational arguments.[7]

We start with a useful definition. We say b *dominates* $1 - b$ at a given point in the execution, if every (non-crashed) node at this point believes b is winning the race, and none of the messages in transit can change this perception.

[7] We had initially hoped there might be some way to simulate linearizable shared objects in our model. Unfortunately, our nodes' lack of information about the network size thwarted standard simulation strategies which typically require nodes to collect messages from a majority of nodes in the network before proceeding to the next step of the simulation.

To formalize this notion we need some notation. In the following, we say *at point t* (or *at t*), with respect to an event t from the message schedule of an execution α, to describe the state of the system immediately after event t (and any associated steps that execute atomically with t) occurs. We also use the notation *in transit at t* to describe messages that have been broadcast but not yet received at every non-crashed receiver at t.

▶ **Definition 1.** Fix an execution α, event t in the corresponding message schedule $msg[\alpha]$, consensus value $b \in \{0, 1\}$, and counter value $c \geq 0$. We say α is (b, c)-*dominated* at t if the following conditions are true:

1. For every node u that is not crashed at t: $\hat{c}_u^{(b)}[t] > c$ and $\hat{c}_u^{(1-b)}[t] \leq c$, where at point t, $\hat{c}_u^{(b)}[t]$ (resp. $\hat{c}_u^{(1-b)}[t]$) is the largest value u has sent or received in a counter message containing consensus value b (resp. $1 - b$). If u has not sent or received any counter messages containing b (resp. $1 - b$), then by default it sets $\hat{c}_u^{(b)}[t] \leftarrow 0$ (resp. $\hat{c}_u^{(1-b)}[t] \leftarrow 0$) in making this comparison.
2. For every message of the form $(counter, id, 1 - b, c', n')$ that is in transit at t: $c' \leq c$.

The following lemma formalizes the intuition that once an execution becomes dominated by a given value, it remains dominated by this value.

▶ **Lemma 2.** *Assume some execution α is (b, c)-dominated at point t. It follows that α is (b, c)-dominated at every t' that comes after t.*

Proof. In this proof, we focus on the suffix of the message schedule $msg[\alpha]$ that begins with event t. For simplicity, we label these events $E_1, E_2, E_3, ...$, with $E_1 = t$. We will prove the lemma by induction on this sequence.

The base case (E_1) follows directly from the lemma statement. For the inductive step, we must show that if α is (b, c)-dominated at point E_i, then it will be dominated at E_{i+1} as well. By the inductive hypothesis, we assume the execution is dominated immediately before E_{i+1} occurs. Therefore, the only way the step is violated is if E_{i+1} transitions the system from dominated to non-dominated status. We consider all possible cases for E_{i+1} and show none of them can cause such a transition.

The first case is if E_{i+1} is a $crash_u$ event for some node u. It is clear that a crash cannot transition a system into non-dominated status.

The second case is if E_{i+1} is a $recv(m)_u$ event for some node u. This event can only transition the system into a non-dominated status if m is a counter message that includes $1 - b$ and a counter $c' > c$. For u to receive this message, however, means that the message was in transit immediately before E_{i+1} occurs. Because we assume the system is dominated at E_i, however, no such message can be in transit at this point (by condition 2 of the domination definition).

The third and final case is if E_{i+1} is a $ack(m)_u$ event for some node u, that is combined with a $bcast(m')_u$ event, where m' is a counter message that includes $1 - b$ and a counter $c' > c$. Consider the values $\hat{c}_u^{(b)}$ and $\hat{c}_u^{(1-b)}$ set by node u early in the steps associated with this $ack(m)_u$ event. By our inductive hypothesis, which tells us that the execution is dominated right before this $ack(m)_u$ event occurs, it must follow that $\hat{c}_u^{(b)} > \hat{c}_u^{(1-b)}$ (as $\hat{c}_u^{(b)} = \hat{c}_u^{(b)}[E_i]$ and $\hat{c}_u^{(1-b)} = \hat{c}_u^{(1-b)}[E_i]$). In the steps that immediately follow, therefore, node u will set $v_u \leftarrow b$. It is therefore impossible for u to then broadcast a counter message with value $v_u = 1 - b$. ◀

To prove agreement, we are left to show that if a node commits to deciding some value b, then it must be the case that b dominates the execution at this point – making it the only possible decision going forward. The following helper lemma, which captures a useful property about counters, will prove crucial for establishing this point.

▶ **Lemma 3.** *Assume event t in the message schedule of execution α is combined with a $bcast(m)_v$, where $m = (counter, id_v, c, b, n_v)$, for some counter $c > 0$. It follows that prior to t in α, every node that is non-crashed at t received a counter message with counter $c - 1$ and value b.*

Proof. Fix some t, α, v and $m = (counter, id_v, c, b, n_v)$, as specified by the lemma statement. Let t' be the first event in α such that at t' some node w has local counter $c_w \geq c$ and value $v_w = b$. We know at least one such event exists as t and v satisfy the above conditions, so the earliest such event, t', is well-defined. Furthermore, because t' must modify local counter and/or consensus values, it must also be an *ack* event.

For the purposes of this argument, let c_w and v_w be w's counter and consensus value, respectively, immediately before t' is scheduled. Similarly, let c'_w and v'_w be these values immediately after t' and its steps complete (i.e., these values at point t'). By assumption: $c'_w \geq c$ and $v'_w = b$. We proceed by studying the possibilities for c_w and v_w and their relationships with c'_w and v'_w.

We begin by considering v_w. We want to argue that $v_w = b$. To see why this is true, assume for contradiction that $v_w = 1 - b$. It follows that early in the steps for t', node w switches its consensus value from $1 - b$ to b. By the definition of the algorithm, it only does this if at this point in the *ack* steps: $\hat{c}_w^{(b)} > \hat{c}_w^{(1-b)} \geq c_w$ (the last term follows because c_w is included in the values considered when defining $c_w^{(1-b)}$). Note, however, that $c_w^{(b)}$ must be less than c. If it was greater than or equal to c, this would imply that a node ended an earlier event with counter $\geq c$ and value b – contradicting our assumption that t' was the earliest such event. If $c_w^{(b)} < c$ and $c_w^{(b)} > c_w$, then w must increase its c_w value during this event. But because $\hat{c}_w^{(b)} > \hat{c}_w^{(1-b)} \geq c_w$, the only allowable change to c_w would be to set it to $\hat{c}_w^{(b)} < c$. This contradicts the assumption that $c'_w \geq c$.

At this checkpoint in our argument we have argued that $v_w = b$. We now consider c_w. If $c_w \geq c$, then w starts t' with a sufficiently big counter – contradicting the assumption that t' is the earliest such event. It follows that $c_w < c$ and w must increase this value during this event.

There are two ways to increase a counter; i.e., the two conditions in the *if/else-if* statement that follows the $newm = \perp$ check. We start with the second condition. If $\max\{\hat{c}_w^{(b)}, \hat{c}_w^{(1-b)}\} > c_w$, then w can set c_w to this maximum. If this maximum is equal to $\hat{c}_w^{(b)}$, then this would imply $\hat{c}_w^{(b)} \geq c$. As argued above, however, it would then follow that a node had a counter $\geq c$ and value b before t'. If this is not true, then $\hat{c}_w^{(1-b)} > c_w^{(b)}$. If this was the case, however, w would have adopted value $1 - b$ earlier in the event, contradicting the assumption that $v'_w = b$.

At this next checkpoint in our argument we have argued that $v_w = b$, $c_w < c$, and w increases c_w to c through the first condition of the *if/else if*; i.e., it must find that $\max\{\hat{c}_w^{(b)}, \hat{c}_w^{(1-b)}\} \leq c_w$ and $m \neq nop$. Because this condition only increases the counter by 1, we can further refine our assumption to $c_w = c - 1$.

To conclude our argument, consider the implications of the $m \neq nop$ component of this condition. It follows that t' is an $ack(m)_w$ for an actual message m. It cannot be the case that m is a *decide* message, as w will not increase its counter on acknowledging a *decide*. Therefore, m is a counter message. Furthermore, because counter and consensus values are not modified after broadcasting a counter message but before receiving its subsequent acknowledgment, we know $m = (counter, id_w, c_w, v_w, *) = (counter, id_w, c - 1, b, *)$ (we replace the network size estimate with a wildcard here as these estimates could change during this period).

Because w has an acknowledgment for this m, by the definition of the model, prior to t': every non-crashed node received a counter message with counter $c - 1$ and consensus value b. This is exactly the claim we are trying to prove. ◀

Our main safety theorem leverages the above two lemmas to establish that committing to decide b means that b dominates the execution. The key idea is that counter values cannot become too stale. By Lemma 3, if some node has a counter c associated with proposal value $1-b$, then all nodes have seen a counter of size at least $c-1$ associated with $1-b$. It follows that if some node thinks b is far ahead, then all nodes must think b is far ahead in the race (i.e., b dominates). Lemma 2 then establishes that this dominance is permanent – making b the only possible decision value going forward.

▶ **Theorem 4.** *The Counter Race Consensus algorithm satisfies validity and agreement.*

Proof. Validity follows directly from the definition of the algorithm. To establish agreement, fix some execution α that includes at least one decision. Let t be the first *ack* event in α that is combined with a broadcast of a *decide* message. We call such a step a *pre-decision* step as it prepares nodes to decide in a later step. Let u be the node at which this *ack* occurs and b be the value it includes in the *decide* message. Because we assume at least one process decides in α, we know t exists. We also know it occurs before any decision.

During the steps associated with t, u sets $newm \leftarrow (decide, b)$. This indicates the following is true: $\hat{c}_u^{(b)} \geq \hat{c}_u^{(1-b)} + 3$. Based on this condition, we establish two claims about the system at t, expressed with respect to the value $\hat{c}_u^{(1-b)}$ during these steps:

- *Claim 1.* The largest counter included with value $1-b$ in a counter message broadcast[8] before t is no more than $\hat{c}_u^{(1-b)} + 1$.

  Assume for contradiction that before t some v broadcast a counter message with value $1-b$ and counter $c > \hat{c}_u^{(1-b)} + 1$. By Lemma 3, it follows that before t every non-crashed node receives a counter message with value $1-b$ and counter $c-1 \geq \hat{c}_u^{(1-b)} + 1$. This set of nodes includes u. This contradicts our assumption that at t the largest counter u has seen associated with $1-b$ is $\hat{c}_u^{(1-b)}$.
- *Claim 2.* Before t, every non-crashed node has sent or received a counter message with value b and counter at least $\hat{c}_u^{(1-b)} + 2$.

  By assumption on the values u has seen at t, we know that before t some node v broadcast a counter message with value b and counter $c \geq \hat{c}_u^{(1-b)} + 3$. By Lemma 3, it follows that before t, every node has sent or received a counter with value b and counter $c-1 \geq \hat{c}_u^{(1-b)} + 2$.

Notice that claim 1 combined with claim 2 implies that the execution is $(b, \hat{c}_u^{(1-b)} + 1)$-dominated before t. By Lemma 2, the execution will remain dominated from this point forward. We assume t was the first pre-decision, and it will lead u to tell other nodes to decide u before doing so itself. Other pre-decision steps might occur, however, before all nodes have received u's preference for b. With this in mind, let t' be any other pre-decision step. Because t' comes after t it will occur in a $(b, \hat{c}_u^{(1-b)} + 1)$-dominated system. This means that during the first steps of t', the node will adopt b as its value (if it has not already done so), meaning it will also promote b.

To conclude, we have shown that once any node reaches a pre-decision step for a value b, then the system is already dominated in favor of b, and therefore b is the only possible decision value going forward. Agreement follows directly. ◀

[8] Notice, in these claims, when we say a message is "broadcast" we only mean that the corresponding *bcast* event occurred. We make no assumption on which nodes have so far received this message.

3.3 Liveness

We now turn our attention liveness. Our goal is to prove the following theorem:

▶ **Theorem 5.** *With high probability, within $O(n^3 \ln n)$ scheduled ack events, every node executing counter race consensus has either crashed, decided, or received a decide message. In the limit, this termination condition occurs with probability* 1*.*

Notice that this theorem does not require a fair schedule. It guarantees its termination criteria (with high probability) after *any* $O(n^3 \ln n)$ scheduled *ack* events, regardless of *which* nodes these events occur at. Once the system arrives at a state in which every node has either crashed, decided, or received a *decide* message, the execution is now univalent (only one decision value is possible going forward), and each non-crashed node u will decide after at most two additional *ack* events at u.[9]

Our liveness proof is longer and more involved than our safety proof. This follows, in part, from the need to introduce multiple technical definitions to help identify the execution fragments sufficiently well-behaved for us to apply our probabilistic arguments. With this in mind, we divide the presentation of our liveness proof into two parts. The first part introduces the main ideas of the analysis and provides a road map of sorts to its component pieces. The second part, which contains the details, can be found in the full paper [42].

3.3.1 Main Ideas

Here we discuss the main ideas of our liveness proof. A core definition used in our analysis is the notion of an *x-run*. Roughly speaking, for a given constant integer $x \geq 2$ and node u, we say an execution fragment β is an x-run for some node u, if it starts and ends with an *ack* event for u, it contains x total *ack* events for u, and no other node has more than x *ack* events interleaved. We deploy a recursive counting argument to establish that an execution fragment β that contains at least $n \cdot x$ total *ack* events, must contain a sub-fragment β' that is an x-run for some node u.

To put this result to use, we focus our attention on $(2c + 1)$-runs, where $c = 6$ is the constant used in the algorithm definition to define the length of a *group* (see Section 3.1 for a reminder of what a group is and how it is used by the algorithm). A straightforward argument establishes that a $(2c + 1)$-run for some node u must contain at least one *complete group* for u – that is, it must contain all c broadcasts of one of u's groups.

Combining these observations, it follows that if we partition an execution into *segments* of length $n \cdot (2c + 1)$, each such segment i contains a $(2c + 1)$-run for some node u_i, and each such run contains a complete group for u_i. We call this complete group the *target group* t_i for segment i (if there are multiple complete groups in the run, choose one arbitrarily to be the target).

These target groups are the core unit to which our subsequent analysis applies. Our goal is to arrive at a target group t_i that is *clean* in the sense that u_i is *active* during the group (i.e., sends its actual values instead of *nop* placeholders), and all broadcasts that arrive at u during this group come from *non-active* nodes (i.e., these received messages contain *nop* placeholders instead of values). If we achieve a *clean* group, then it is not hard to show that u_i will advance its counter at least k ahead of all other counters, pushing all other nodes into the termination criteria guaranteed by Theorem 5.

[9] In the case where u receives a *decide* message, the first *ack* might correspond to the message it was broadcasting when the *decide* arrived, and the second *ack* corresponds to the *decide* message that u itself will then broadcast. During this second *ack*, u will decide and halt.

To prove clean groups are sufficiently likely, our analysis must overcome two issues. The first issue concerns network size estimations. Fix some target group t_i. Let P_i be the nodes from which u_i receives at least one message during t_i. If all of these nodes have a network size estimate of at least $n_i = |P_i|$ at the start of t_i, we say the group is *calibrated*. We prove that if t_i is calibrated, then it is clean with a probability in $\Omega(1/n)$.

The key, therefore, is proving most target groups are calibrated. To do so, we note that if some t_i is not calibrated, it means at least one node used an estimate strictly less than n_i when it probabilistically defined *active* at the beginning of this group. During this group, however, all nodes will receive broadcasts from at least n_i unique nodes, increasing all network estimates to size at least n_i.[10] Therefore, each target group that fails to be calibrated increases the minimum network size estimate in the system by at least 1. It follows that at most n target groups can be non-calibrated.

The second issue concerns probabilistic dependencies. Let E_i be the event that target group t_i is clean and E_j be the event that some other target group t_j is clean. Notice that E_i and E_j are not necessarily independent. If a node u has a group that overlaps both t_i and t_j, then its probabilistic decision about whether or not to be active in this group impacts the potential cleanliness of both t_i and t_j.

Our analysis tackles these dependencies by identifying a subset of target groups that are pairwise independent. To do so, roughly speaking, we process our target groups in order. Starting with the first target group, we mark as unavailable any future target group that overlaps this first group (in the sense described above). We then proceed until we arrive at the next target group *not* marked unavailable and repeat the process. Each available target group marks at most $O(n)$ future groups as unavailable. Therefore, given a sufficiently large set T of target groups, we can identify a subset T', with a size in $\Omega(|T|/n)$, such that all groups in T' are pairwise independent.

We can now pull together these pieces to arrive at our main liveness complexity claim. Consider the first $O(n^3 \ln n)$ *ack* events in an execution. We can divide these into $O(n^2 \ln n)$ segments of length $(2c+1)n \in \Theta(n)$. We now consider the target groups defined by these segments. By our above argument, there is a subset T' of these groups, where $|T'| \in \Omega(n \ln n)$, and all target groups in T' are mutually independent. At most n of these remaining target groups are not calibrated. If we discard these, we are left with a slightly smaller set, of size still $\Omega(n \ln n)$, that contains only calibrated and pairwise independent target groups.

We argued that each calibrated group has a probability in $\Omega(1/n)$ of being clean. Leveraging the independence between our identified groups, a standard concentration analysis establishes with high probability in n that at least one of these $\Omega(n/\ln n)$ groups is clean – satisfying the Theorem statement.

3.4 Removing the Assumption of Unique IDs

The consensus algorithm described in this section assumes unique IDs. We now show how to eliminate this assumption by describing a strategy that generates unique IDs w.h.p., and discuss how to use this as a subroutine in our consensus algorithm.

We make use of a simple tiebreaking mechanism as follows: Each node u proceeds by iteratively extending a (local) random bit string that eventually becomes unique among the nodes. Initially, u broadcasts bit b_1, which is initialized to 1 (at all nodes), and each time u

[10] This summary is eliding some subtle details tackled in the full analysis concerning which broadcasts are guaranteed to be received during a target group. But these details are not important for understanding the main logic of this argument.

samples a new bit b, it appends b to its current string and broadcasts the result. For instance, suppose that u's most recently broadcast bit string is $b_1 \dots b_i$. Upon receiving $ack(b_1 \dots b_i)$, node u checks if it has received a message identical to $b_1 \dots b_i$. If it did not receive such a message, then u adopts $b_1 \dots b_i$ as its ID and stops. Otherwise, some distinct node must have sampled the same sequence of bits as u and, in this case, the ID $b_1 \dots b_i$ is considered to be already taken. (Note that nodes do not take receive events for their own broadcasts.) Node u continues by sampling its $(i+1)$-th bit b_{i+1} uniformly at random, and then broadcasts the string $b_1 \dots b_i b_{i+1}$, and so forth. In the full paper [42], we prove the following result and describe how to combine it with our consensus algorithm:

▶ **Theorem 6.** *Consider an execution α of the tiebreaking algorithm. Let t_u be an event in the message schedule $msg[\alpha]$ such that node u is scheduled for $\Omega(\log n)$ ack events before t_u. Then, for each correct node u, it holds that u has a unique ID of $O(\log n)$ bits with high probability at t_u.*

4 Almost-Everywhere Agreement

In the previous section, we showed how to solve consensus in $O(n^3 \log n)$ events. Here we show how to improve this bound by a near linear factor by loosening the agreement guarantees. In more detail, we consider a weaker variant of consensus, introduced in [18], called *almost-everywhere agreement*. This variation relaxes the agreement property of consensus such that $o(n)$ nodes are allowed to decide on conflicting values so long as the remaining nodes all decide the same value. For many problems that use consensus as a subroutine, this relaxed agreement property is sufficient.

In more detail, we present an algorithm for solving almost-everywhere agreement in the abstract MAC layer model when nodes start with arbitrary (not necessarily binary) input values. The algorithm consists of two phases. We present the pseudo code in the full paper [42].

Phase 1. In this phase, nodes try to obtain an estimate of the network size by performing local coin flipping experiments. Each node u records the number of times that its coin comes up tails before observing the first heads in a variable X. Then, u broadcasts its value of X once, and each node updates X to the highest outcome that it has seen until it receives the *ack* for its broadcast. In our analysis, we show that, by the end of Phase 1, variable X is an approximation of $\log_2(n)$ with an additive $O(\log \log n)$ term, for all nodes in a large set called EST, and hence $N := 2^X$ is a good approximation of the network size n for any node in EST.

Phase 2. Next, we use X and N as parameters of a randomly rotating leader election procedure. Each node decides after $T = \Theta(N \log^3(N) \log \log(N))$ *rounds*. (Note that due to the asynchronous nature of the abstract MAC layer model, different nodes might be executing in different rounds at the same point in time.) We now describe the sequence of steps comprising a round in more detail: A node u becomes active with probability $1/N_u$ at the start of each round.[11] If it is active, then u samples a random rank ρ from a range polynomial in X_u, and broadcasts a message $\langle r, \rho, val \rangle$ where val refers to its current consensus input value. To ensure that the scheduler cannot derive any information about

[11] We use the convention N_u when referring to the local variable N of a specific node u.

whether a node is active in a round, inactive nodes simply broadcast a dummy message with infinite rank. While an (active or inactive) node v waits for its *ack* for round r, it keeps track of all received messages and defers processing of a message sent by a node in some round $r' > r$ until the event in which v itself starts round r'. On the other hand, if a received message was sent in $r' < r$, then v simply discards that late message as it has already completed r'. Node v uses the information of messages originating from the same round r to update its consensus input value, if it receives such a message from an active node that has chosen a smaller rank than its own. (Recall that inactive nodes have infinite rank.) After v has finished processing the received messages, it moves on the next round.

We first provide some intuition why it is insufficient to focus on a round r where the "earliest" node is also active: Ideally, we want the node w_1 that is the first to receive its *ack* for round r to be active *and* to have the smallest rank among all active nodes in round r, as this will force all other (not-yet decided) nodes to adopt w_1's value when receiving their own round r *ack*, ensuring a.e. agreement. However, it is possible that w_1 and also the node w_2 that receives its round r *ack* right after w_1, are among the few nodes that ended up with a small (possibly constant) value of X after Phase 1. We cannot use the size of EST to reason about this probability, as some nodes are much likelier to be in EST than others, depending on the schedule of events in Phase 1. In that case, it could happen that both w_1 and w_2 become active and choose a rank of 1. Note that it is possible that the receive steps of their broadcasts are scheduled such that roughly half of the nodes receive w_1's message before w_2's message, while the other half receive w_2's message first. If w_1 and w_2 have distinct consensus input values, then it can happen that both consensus values gain large support in the network as a result.

To avoid this pitfall, we focus on a set of rounds where all nodes *not* in EST have already terminated Phase 2 (and possibly decided on a wrong value): from that point onwards, only nodes with sufficiently large values of X and N keep trying to become active. We can show that every node in EST has a probability of at least $\Omega(1/(n \log n))$ to become active and a probability of $\Omega(1/\log n)$ to have chosen the smallest rank among all nodes that are active in the same round. Thus, when considering a sufficiently large set of rounds, we can show that the event, where the first node in EST that receives its *ack* in round r becomes active and also chooses a rank smaller than the rank of any other node active active in the same round, happens with probability $1 - o(1)$.

In the full paper [42], we formalize the above discussion by proving the following main theorem regarding this algorithm:

▶ **Theorem 7.** *With high probability, the following two properties are true of our almost-everywhere consensus algorithm: (1) within $O(n^2 \log^4 n \cdot \log \log n)$ scheduled ack events, every node has either crashed, decided, or will decided after it is next scheduled; (b) all but at most $o(n)$ nodes that decide, decide the same value.*

5 Lower Bound

We conclude our investigation by showing a separation between the abstract MAC layer model and the related asynchronous message passing model. In more detail, we prove below that fault-tolerant consensus with constant success probability is impossible in a variation of the asynchronous message passing model where nodes are provided only a constant-fraction approximation of the network size and communicate using (blind) broadcast. This bounds holds even if we assume no crashes and provide nodes unique ids from a small set. Notice, in the abstract MAC layer model, we solve consensus with broadcast under the harsher

Algorithm 3 Almost-everywhere agreement in the abstract MAC layer model. Code for node u.

```
1: val ← consensus input value
2:                                                              ▷ Phase 1
3: initialize X ← 0; R ← ∅
4: while flip_coin() = heads do
5:     X ← X + 1
6: bcast(X)
7: while waiting for ack do
8:     add received messages to R
9: X ← max(R ∪ {X})
10: N ← 2^X
11:                                                             ▷ Phase 2
12: T ← ⌈cN log^3(N) log log(N)⌉, where c is a sufficiently large constant.
13: initialize array of sets R[1], ..., R[T] ← ∅
14: for i ← 1, ..., T do                                        ▷ Start of round i at u
15:     u becomes active with probability 1/N
16:     if u is active then
17:         ρ ← unif. at random sampled integer from [1, X^4]
18:     else
19:         ρ ← ∞
20:     bcast(⟨i, ρ, val⟩)
21:     while waiting for ack do
22:         add received messages to R[i]
23:     for each message m = ⟨i', ρ', val'⟩ ∈ R[i] do
24:         if i' = i and ρ' < ρ then          ▷ Received message from node with smaller rank
25:             val ← val'
26:         else if i' > i then                ▷ Received message from node active in future round
27:             add m to R[i']
28:         else
29:             discard message m
30: decide on val
```

constraints of no network size information, no ids, and crash failures. The difference is the fact that the broadcast primitive in the abstract MAC layer model includes an acknowledgment. This acknowledgment is therefore revealed to be the crucial element of the our model that allows algorithms to overcome lack of network information. We note that this bound is a generalization of the result from [1], which proved deterministic consensus was impossible under these constraints. In the full paper [42], we show that, for any given randomized algorithm we can construct scenarios that are indistinguishable for the nodes, thus causing conflicting decisions.

▶ **Theorem 8.** *Consider an asynchronous network of n nodes that communicate by broadcast and suppose that nodes are unaware of the network size n, but have knowledge of an integer that is guaranteed to be a 2-approximation of n. No randomized algorithm can solve binary consensus with a probability of success of at least $1 - \epsilon$, for any constant $\epsilon < 2 - \sqrt{3}$. This holds even if nodes have unique identifiers chosen from a range of size at least $2n$ and all nodes are correct.*

References

1 Mohssen Abboud, Carole Delporte-Gallet, and Hugues Fauconnier. Agreement without knowing everybody: a first step to dynamicity. In *Proceedings of the International Conference on New Technologies in Distributed Systems*, 2008.

2 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000.

3 Eduardo AP Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *Proceedings of the International Conference on the Principles of Distributed Systems*. Springer, 2008.

4 Khaled Alekeish and Paul Ezhilchelvan. Consensus in sparse, mobile ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):467–474, 2012.

5 James Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1):16–39, 2002.

6 Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.

7 John Augustine, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Distributed agreement in dynamic peer-to-peer networks. *J. Comput. Syst. Sci.*, 81(7):1088–1109, 2015. doi:10.1016/j.jcss.2014.10.005.

8 R. Bar-Yehuda, O. Goldreich, and A. Itai. On the Time Complexity of Broadcast in Radio Networks: an Exponential Gap Between Determinism and Randomization. In *Proceedings of the ACM Conference on Distributed Computing*, 1987.

9 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the ACM Conference on Distributed Computing*, pages 27–30. ACM, 1983.

10 François Bonnet and Michel Raynal. Anonymous Asynchronous Systems: the Case of Failure Detectors. In *Proceedings of the International Conference on Distributed Computing*, 2010.

11 David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In *ADHOC-NOW*, 2004.

12 Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the ACM Conference on Distributed Computing*, 1996.

13 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

14 Alejandro Cornejo, Nancy Lynch, Saira Viqar, and Jennifer L Welch. Neighbor Discovery in Mobile Ad Hoc Networks Using an Abstract MAC Layer. In *Proceedings of the Annual Allerton Conference on Communication, Control, and Computing*, 2009.

15 Alejandro Cornejo, Saira Viqar, and Jennifer L Welch. Reliable Neighbor Discovery for Mobile Ad Hoc Networks. *Ad Hoc Networks*, 12:259–277, 2014.

16 A. Czumaj and W. Rytter. Broadcasting Algorithms in Radio Networks with Unknown Topology. *Journal of Algorithms*, 60:115–143, 2006.

17 Sebastian Daum, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Broadcast in the Ad Hoc SINR Model. In *Proceedings of the International Conference on Distributed Computing*, 2013.

18 Cynthia Dwork, David Peleg, Nicholas Pippenger, and Eli Upfal. Fault tolerance in networks of bounded degree. *SIAM Journal on Computing*, 17(5):975–988, 1988.

19 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.

20 L. Gasieniec, D. Peleg, and Q. Xin. Faster Communication in Known Topology Radio Networks. *Distributed Computing*, 19(4):289–300, 2007.

21 O. Goussevskaia, R. Wattenhofer, M.M. Halldorsson, and E. Welzl. Capacity of Arbitrary Wireless Networks. In *Proceedings of the IEEE International Conference on Computer Communications*, 2009.

22 Fabiola Greve and Sebastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

23 Rachid Guerraoui, Michel Hurfinn, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. *Advances in Distributed Systems, Lecture Notes in Computer Science*, 1752:33–47, 2000.

24 Rachid Guerraoui and Andre Schiper. Consensus: the big misunderstanding [distributed fault tolerant systems]. In *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997.

25 Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.

26 Magnus M. Halldorsson and Pradipta Mitra. Wireless Connectivity and Capacity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2012.

27 Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rozanski, and Grzegorz Stachowiak. Distributed Randomized Broadcasting in Wireless Networks under the SINR Model. In *Proceedings of the International Conference on Distributed Computing*, 2013.

28 Tomasz Jurdziński and Grzegorz Stachowiak. Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In *Algorithms and Computation*, pages 535–549. Springer, 2002.

29 Majid Khabbazian, Fabian Kuhn, Dariusz Kowalski, and Nancy Lynch. Decomposing Broadcast Algorithms Using Abstract MAC Layers. In *Proceedings of the International Workshop on the Foundations of Mobile Computing*, 2010.

30 Majid Khabbazian, Fabian Kuhn, Nancy Lynch, Muriel Medard, and Ali ParandehGheibi. MAC Design for Analog Network Coding. In *Proceedings of the International Workshop on the Foundations of Mobile Computing*, 2011.

31 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 87–98. IEEE, 2006.

32 D.R. Kowalski and A. Pelc. Broadcasting in Undirected Ad Hoc Radio Networks. *Distributed Computing*, 18(1):43–57, 2005.

33 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The Abstract MAC Layer. In *Proceedings of the International Conference on Distributed Computing*, 2009.

34 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The Abstract MAC Layer. *Distributed Computing*, 24(3-4):187–206, 2011.

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

36 Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

37 Thomas Moscibroda. The Worst-Case Capacity of Wireless Sensor Networks. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2007.

38 Thomas Moscibroda and Roger Wattenhofer. Maximal Independent Sets in Radio Networks. In *Proceedings of the ACM Conference on Distributed Computing*, 2005.

39 Thomas Moscibroda and Roger Wattenhofer. The Complexity of Connectivity in Wireless Networks. In *Proceedings of the IEEE International Conference on Computer Communications*, 2006.

40 Achour Mostefaoui and Michel Raynal. Solving consensus using Chandra-Touegs unreliable failure detectors. *Lecture Notes in Computer Science*, 1693:49–63, 1999.

41 Calvin Newport. Consensus with an Abstract MAC Layer. In *Proceedings of the ACM Conference on Distributed Computing*, 2014.

42 Calvin Newport and Peter Robinson. Fault-Tolerant Consensus with an Abstract MAC Layer. Technical report, `https://www.cas.mcmaster.ca/robinson/random-aml.pdf`, 2018.

43 Eric Ruppert. The Anonymous Consensus Hierarchy and Naming Problems. In *Proceedings of the International Conference on Principles of Distributed Systems*, 2007.

44 Andre Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

45 Einar W Vollset and Paul D Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in MANETs. In *IEEE Symposium on Reliable Distributed Systems*, 2005.

46 Weigang Wu, Jiannong Cao, and Michel Raynal. Eventual clusterer: A modular approach to designing hierarchical consensus protocols in manets. *IEEE Transactions onParallel and Distributed Systems*, 20(6):753–765, 2009.