Consensus with an Abstract MAC Layer

Calvin Newport
Georgetown University
cnewport@cs.georgetown.edu

Abstract

In this paper, we study distributed consensus in the radio network setting. We produce new upper and lower bounds for this problem in an abstract MAC layer model that captures the key guarantees provided by most wireless MAC layers. In more detail, we first generalize the well-known impossibility of deterministic consensus with a single crash failure [15] from the asynchronous message passing model to our wireless setting. Proceeding under the assumption of no faults, we then investigate the amount of network knowledge required to solve consensus in our model—an important question given that these networks are often deployed in an ad hoc manner. We prove consensus is impossible without unique ids or without knowledge of network size (in multihop topologies). We also prove a lower bound on optimal time complexity. We then match these lower bounds with a pair of new deterministic consensus algorithms—one for single hop topologies and one for multihop topologies—providing a comprehensive characterization of the consensus problem in the wireless setting. From a theoretical perspective, our results shed new insight into the role of network information and the power of MAC layer abstractions in solving distributed consensus. From a practical perspective, given the level of abstraction used by our model, our upper bounds can be easily implemented in real wireless devices on existing MAC layers while preserving their correctness guarantees—facilitating the development of wireless distributed systems.

1 Introduction

Consensus provides a fundamental building block for developing reliable distributed systems [20, 19, 21]. Motivated by the increasing interest in *wireless* distributed systems, in this paper we prove new upper and lower bounds for the consensus problem in wireless networks.

The Abstract MAC Layer. Consensus bounds are dependent on the model in which they are established. Accordingly, we must take care in selecting our model for studying the wireless version of this problem. Most existing work on distributed algorithms for wireless networks assumes low-level synchronous models that require algorithms to deal directly with link-layer issues such as signal fading and channel contention. Some of these models use topology graphs to determine message behavior (c.f., [6, 24, 27, 34, 13, 16]) while others use signal strength calculations (c.f., [35, 33, 17, 22, 23, 14]). These models are well-suited for asking basic science questions about the capabilities of wireless communication. They are not necessarily appropriate, however, for developing algorithms meant for deployment, as real wireless systems typically require an algorithm to operate on top of a general-purpose MAC layer which is hard to bypass and enables many key network functions such as managing co-existence.

Motivated by this reality, in this paper we adopt the *abstract MAC layer* approach [29], in which we model the basic guarantees provided by most existing wireless MAC layers—if you broadcast a message it will eventually be delivered with acknowledgment to nearby nodes in the network—but leverages a non-deterministic message scheduler to allow for unpredictability—there is no bound on when messages are delivered or in what order. The goal with this approach is to describe and analyze algorithms at a level of abstraction that makes it easy to subsequently implement theory results in real systems while still preserving their formally analyzed properties. (See Section 2 for a detailed model definition and motivation.)

Results. We begin with lower bounds. In Section 3.1, we generalize the oft-cited result on the impossibility of deterministic consensus with a single process failure [15] from the asynchronous message passing model

to our abstract MAC layer model. (See Section 2 for details on how these two models differ.) The main difficulty in this generalization is the new assumption in our model that senders receive acknowledgments at some point after a given broadcast completes. To overcome this difficulty, we are forced to restrict our valency definitions to focus on a restricted class of schedulers.

Having established this impossibility, we proceed in this paper assuming no crash failures. Noting that wireless network deployments are often ad hoc, we next focus on determining how much *a priori* information about the network is required to solve deterministic consensus in our model. We start, in Section 3.2, by proving that consensus is impossible without unique ids, even if nodes know the size and diameter of the network. We then prove, in Section 3.3, that even with unique ids (and knowledge of the diameter), consensus is impossible in multihop networks if nodes do not know the network size. Finally, we prove that any solution to consensus in our model requires $\Omega(D \cdot F_{ack})$ time, where D is the diameter of the underlying network topology and F_{ack} is the maximum message delivery delay (a value unknown to the nodes in the network). All three bounds leverage partitioning arguments that rely on carefully-constructed worst-case network topologies and message scheduler behavior for which the relevant network knowledge assumptions do not break symmetry.

We then turn our attention to matching these lower bounds with a pair of new deterministic consensus algorithms. We begin, in Section 4.1, with a new algorithm that guarantees to solve consensus in single hop networks in an optimal $O(F_{ack})$ time, even without advance knowledge of the network size or participants (this opens up a gap with the asynchronous message passing model, where consensus is impossible under such assumptions [1]). This algorithm uses a two-phase structure. The key insight is that nodes wait to decide after their second phase broadcast until they have also heard this broadcast from a set of important *witnesses*.

We then present, in Section 4.2, the *wireless PAXOS* (wPAXOS) algorithm, which guarantees to solve consensus in multihop topologies of diameter D in an optimal $O(D \cdot F_{ack})$ time. This algorithm assumes unique ids and knowledge of n (as required by our lower bounds[1]), but no other advance knowledge of the network or participants. The wPAXOS algorithm combines the high-level logic of the PAXOS consensus algorithm [30] with a collection of support services that efficiently disseminate proposals and aggregate responses. We note that if the PAXOS (or similar consensus algorithm) logic is combined with a basic flooding algorithm, the result would be a $O(n \cdot F_{ack})$ time complexity, as bottlenecks are possible where $\Omega(n)$ value and id pairs must be sent by a single node only able to fit $O(1)$ such pairs in each message. To reduce this time complexity to an optimal $O(D \cdot F_{ack})$, we implement eventually stable shortest-path routing trees and show they allow fast aggregation once stabilized, and preserve safety at all times. These stabilizing support services and their analysis represent the main contribution of this algorithm. One could, for example, replace the PAXOS logic working with these services with something simpler (since we have unique ids and knowledge of n, and no crash failures, we could, for example, simply gather all values at all nodes). We choose PAXOS mainly for performance reasons, as it only depends on a majority nodes to make progress, and is therefore not slowed if a small portion of the network is delayed.

Related Work. Consensus provides a fundamental building block for reliable distributed computing [20, 19, 21]. It is particularly well-studied in asynchronous models [30, 39, 36, 2], where deterministic solutions are impossible with even a single crash failure [15]. Most existing distributed algorithm results for the wireless setting assume low-level models. Though consensus has been studied in such models (e.g., [10]), most efforts in the low-level setting focus on reliable communication problems such as broadcast (see [37] for a good survey). The abstract MAC layer approach to modeling wireless networks is introduced in [28] (later expanded to a journal version [29]), and has been subsequently used to study several different problems [11, 25, 26, 12]. This paper, however, is the first to consider consensus in the abstract MAC layer context.

Other researchers have also studied consensus in wireless networks at higher levels of abstraction. Vollset and Ezhilchelvan [40], 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

[1]Our algorithm still works even if provided only *good enough* knowledge of n to recognize a majority. This does not contradict our lower bound as the lower bound assumes *no* knowledge of n.

allowable channel changes. Wu et al. [41] use the standard asynchronous message passing model (with unreliable failure detectors [9]) 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.

Finally, we note that a key focus in this paper is understanding the importance of network information in solving consensus, a topic previously studied in the classical models. Ruppert [38], and Bonnet and Raynal [7], 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. [5] describe consensus solutions for shared memory systems without failures or unique ids. In this paper, by contrast, we prove consensus impossible without failures or unique ids. These results do not contradict, however, as we assume multihop message passing-style networks. A series of papers [8, 18, 3], starting with the work of Cavin et al. [8], 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. Results on the CUPs problem focus on the structure of the knowledge from such detectors required for consensus (e.g., if we create a graph with a directed edge indicating participant knowledge, then the resulting graph must satisfy certain connectivity properties). Closer to our own model is the work of Abboud et al. [1], which studies single hop networks in which participants are *a priori* unknown, but nodes do have a reliable broadcast primitive. They prove consensus is impossible in single hop networks under these assumptions without knowledge of network size. In Section 4.1, we describe an algorithm in our model that *does* solve consensus under these assumptions: opening a gap between these two models.

2 Model and Problem

For simplicity, in the following we sometimes call our model *the* abstract MAC layer model. We emphasize, however, that there is no single abstract MAC layer model, but instead many variants that share the same basic assumptions of acknowledged local broadcast and an arbitrary scheduler. The major differences between our model and the standard asynchronous message passing model are that: (1) we assume local broadcast instead of point-to-point communication; (2) senders receive an acknowledgment at some point after their broadcast completes (this acknowledgment captures the time at which the underlying link layer is done broadcasting its current message; e.g., after its slot in a TDMA schedule arrives or its CSMA algorithm finally detected a clear channel); and (3) we care about assumptions regarding network information knowledge as wireless networks are often deployed in an ad hoc manner where such information may be unknown to nodes.

Model Details. To formalize our abstract MAC layer model, fix a graph $G = (V, E)$, with the set V describing the $|V| = n$ wireless devices in the network (called *nodes* in the following), and the edges in E describing nodes within reliable communication range. In this model, nodes communicate with a local reliable (but not necessarily atomic[2]) broadcast primitive that guarantees to eventually deliver messages to a node's neighbors in G. At some point after a broadcast completes (see below), a node receives an ack. If a node attempts to broadcast additional messages before receiving an ack for the current message, those extra messages are discarded. To formalize the message delivery guarantees, fix some execution α of a deterministic algorithm in our model. To simplify definitions, assume w.l.o.g. that messages are unique. Let π be the event in α where u calls *broadcast*(m), and π' be the subsequent ack returned to u. Our abstract MAC layer model guarantees that in the interval from π to π' in α, every non-faulty neighbor of u in G receives m, and these are the only receive events for m in α (this is where we leverage message uniqueness in our definition).

We associate each message scheduler with an unknown (to the nodes) but finite value F_{ack} that bounds the maximum delay it is allowed between a broadcast and a corresponding acknowledgment. This property induces some notion of fairness: the scheduler must eventually allow each broadcast to finish. To simplify timing, we assume local non-communication steps take no time. That is, all non-determinism is captured in the message receive and ack scheduling. We note that in some definitions of abstract MAC layer models

[2]Local broadcast in wireless networks is not necessarily an atomic operation, as effects such as the hidden terminal problem might lead some neighbors of a sender to receive a message before other neighbors.

(see [29]), a second timing parameter, F_{prog}, is introduced to bound the time for a node to receive *some* message when one or more neighbors are broadcasting. We omit this parameter in this study as it is used mainly for refining time complexity analysis, while we are concerned here more with safety properties. Refining our upper bound results in a model that includes this second parameter remains useful future work. We also note that some definitions of the abstract MAC layer assume a second topology graph consisting of *unreliable* links that sometimes deliver messages and sometimes do not. We omit this second graph in this analysis, which strengthens our lower bounds. Optimizing our multihop upper bound to work in the presence of such links, however, is left an open question.

In some results that follow, we consider *crash* failures (a node halts for the remainder of the execution). The decision to crash a node and the timing of the crash is determined by the scheduler and can happen in the middle of a broadcast (i.e., after some neighbors have received the message but not all). We call a node *non-faulty* (equiv. *correct*) with respect to a given execution if it does not crash. For our upper bounds, we restrict the message size to contain at most a constant number of unique ids. For a given topology graph G, we use D to describe its diameter. Finally, for integer $i > 0$, let $[i] = \{1, 2, ..., i\}$.

The Consensus Problem. To better understand the power of our abstract MAC layer model we explore upper and lower bounds for the standard binary consensus problem. In more detail, each node begins an execution with an initial value from $\{0, 1\}$. Every node has the ability to perform a single irrevocable *decide* action for a value in $\{0, 1\}$. 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 v, then some node had v as its initial value; and (3) *termination*: every non-faulty process eventually decides. By focusing on binary consensus, as oppose to the more general definition that assumes an arbitrary value set, we strengthen the lower bounds that form the core of this paper. Generalizing our upper bounds to the general case in an efficient manner (e.g., a solution more efficient than agreeing on the bits of a general value, one by one, using binary consensus) is non-trivial and remains an open problem.

3 Lower Bounds

We begin by exploring the fundamental limits of our abstract MAC layer model with respect to the consensus problem. In Section 4, we provide matching upper bounds. In the following, we defer some proofs to the appendix for the sake of clarity and concision.

3.1 Consensus is Impossible with Crash Failures

In this section we prove consensus is impossible in our model in the presence of even a single crash failure. To achieve the strongest possible bound we assume a clique topology. Our proof generalizes the FLP [15] result to hold in our stronger setting where nodes now have acknowledgments.[3]

Preliminaries. For this proof, assume w.l.o.g. that nodes always send messages; i.e., on receiving an ack for their current message they immediately begin sending a new message. We define a *step* of a node u to be either: (a) a node $v \neq u$ receiving u's current message; or (b) u receiving an ack for its current message (at which point its algorithm advances to sending a new message). We call a step of type (a) from above *valid* with respect to the execution so far if the node v receiving u's message has not previously received that message *and* all non-crashed nodes smaller than v (by some fixed but arbitrary ordering) have already received u's message. We call a step of type (b) *valid* with respect to the execution so far if every non-crashed neighbor of u has received its current message in a previous step. When we consider executions that consist only of valid steps we are, in effect, restricting our attention to a particular type of well-behaved message scheduler.

[3]The version of the FLP proof that we directly generalize here is the cleaned up and condensed version that appeared in the subsequent textbook of Lynch [32].

We call an execution fragment (equiv. prefix) α of a consensus algorithm *bivalent* if there is some extension of valid steps that leads to nodes deciding 0, and some extension of valid steps that leads to nodes deciding 1. By contrast, we call an execution α *univalent* if every extension of valid steps from α that leads to a decision leads to the same decision.[4] If this decision is 0 (resp. 1), we also say that α is 0-*valent* (resp. 1-*valent*). In the following, we use the notation $\alpha \cdot s$, for execution fragment α and step s, to describe the extension of α by s.

Result. Fix some algorithm $\mathcal{A}$. Assume for the sake of contradiction that $\mathcal{A}$ guarantees to solve consensus in this setting with up to 1 crash failure. The key to generalizing the FLP impossibility to our model is the following lemma, which reproves the main argument of this classical result in a new way that leverages our model-specific constraints.

Lemma 3.1. *Fix some bivalent execution fragment α of $\mathcal{A}$ and some process u. There exists a finite extension α' of α such that $\alpha' \cdot s_u$ is bivalent, where s_u is a valid step of u with respect to α'.*

Proof. Assume for contradiction that this property does not hold for some α and u. It follows that for every finite extension α' of α of valid steps, if we extend α' by a single additional valid step of u, the execution is univalent. Let s_u be the next valid step for u after α (by our definition of valid, s_u is well-defined). We start by considering $\alpha \cdot s_u$. By assumption, this fragment is univalent. Assume, w.l.o.g. that $\alpha \cdot s_u$ is 0-valent (the argument below is symmetric for the case where $\alpha \cdot s_u$ is instead 1-valent). Because α is bivalent, however, there is some other extension α'' consisting of valid steps that is 1-valent.

We now move step by step in α'', starting from α, until our growing fragment becomes 1-valent. Assume there are $k \geq 1$ steps in this extension. Label the k intermediate execution fragments from α to the α'': $\alpha_1, \alpha_2, ..., \alpha_k$, where α_k is the where the fragment becomes 1-valent. To simplify notation, let $\alpha_0 = \alpha$. For $0 < i < k$, we know α_i is bivalent, so, by our contradiction assumption, that step between α_{i-1} and α_i cannot be s_u. Let s^* be the step between α_{k-1} and α_k. (Notice that it *is* possible that $s^* = s_u$, as the execution is no longer bivalent after this final step.)

By our contradiction assumption, we know that for each $i \in \{0, ..., k-1\}$, $\alpha_i \cdot s_u$ is univalent. We also know that $\alpha_0 \cdot s_u$ is 0-valent. It follows that there must exist some $\hat{i} \in \{0, ..., k-1\}$, such that $\alpha_{\hat{i}} \cdot s_u$ is 0-valent and $\alpha_{\hat{i}} \cdot s_v \cdot s_u$ is 1-valent, where s_v is the next valid step of some node $v \neq u$. Notice this holds whether or not $s^* = s_u$ (if $s^* = s_u$ then α_k is a fragment ending with s_u that we know to be 1-valent, otherwise, $\alpha_k \cdot s_u$ is this fragment). We have found a fragment α^*, therefore, where $\alpha^* \cdot s_u$ is 0-valent but $\alpha^* \cdot s_v \cdot s_u$ is 1-valent. We now perform a case analysis on s_v and s_u to show that all possible cases lead to a contradiction. In the following, to simplify notation, let $\beta_0 = \alpha^* \cdot s_u$ and $\beta_1 = \alpha^* \cdot s_v \cdot s_u$.

Case 1*:* Both steps affect the same node w. It is possible that w is u or v (e.g., if s_u is an acknowledgment and s_v is u receiving v's message), it is also possible that w is not u or v (e.g., if s_u and s_v are both receives at some third node w). We note that w (and only w) can distinguish between β_0 and β_1. Imagine, however, that we extend β_1 such that every node *except for* w keeps taking valid steps. All non-w nodes must eventually decide, as this is equivalent to a fair execution where w crashes after β_1, and w is the only node to crash—a setting where termination demands decision. By our valency assumption, these nodes must decide 1. Now imagine that we extend β_0 with the exact same steps. For all non-w nodes these two executions are indistinguishable, so they will once again decide 1. We assumed, however, that β_0 was 0-valent: a contradiction.

Case 2*:* The steps affect two different nodes. In this case, it is clear that no node can distinguish between $\beta_0 \cdot s_v$ and β_1. We can, therefore, apply the same style of indistinguishability argument as in case 1, except in this case we can allow all nodes to continue to take steps. □

We now leverage Lemma 3.1 to prove our main theorem.

[4]Notice, not every extension need lead to a decision. If, for example, the extension does not give steps to two or more nodes, than this is equivalent to two or more nodes crashing—a circumstance for which we do not expect a 1-fault tolerant algorithm to necessarily terminate.

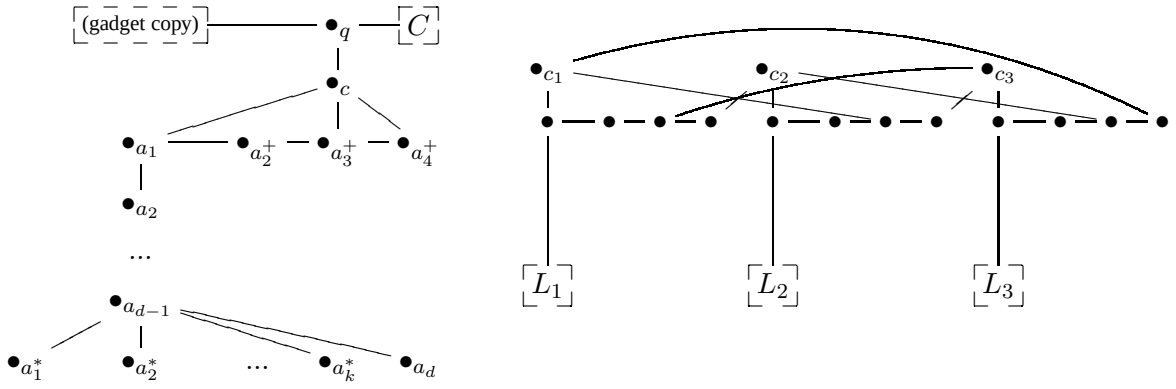

Figure 1: In **Network A** (left) the a nodes plus c combine to comprise a *gadget*. The bridge node q is connected to two copies of this gadget at their c nodes. It is also connected to all nodes in a clique C used to adjust the total network size. In **Network B** (right) the sub-graphs L_1, L_2, and L_3, are each a copy of the sub-graph of the gadget of Network A consisting of nodes at a_2 and below in the diagram (i.e., nodes labelled a_i for $i > 1$, as well as the a_j^* nodes.) L_1, L_2, and L_3, in other words, connect to the a_1 node of the Network A gadget.

Theorem 3.2. *There does not exist a deterministic algorithm that guarantees to solve consensus in a single hop network in our abstract MAC layer model with a single crash failure.*

Proof. Assume for contradiction such an algorithm exists. Call it $\mathcal{A}$. Using the standard argument we first establish the existence of a bivalent initial configuration of $\mathcal{A}$ (e.g., Lemma 2 from [15]). Starting from this configuration, we keep applying Lemma 3.1, rotating through all n nodes in round robin order, to extend the execution in a way that keeps it bivalent. Because we rotate through all nodes when applying Lemma 3.1, the resulting execution is fair in the sense that all nodes keep taking steps. The termination property of consensus requires that nodes eventually decide. By agreement when any node decides the execution becomes univalent. By Lemma 3.1, however, our execution remains bivalent, so no node must ever decide. This violates termination and therefore contradicts the assumption that $\mathcal{A}$ solves consensus. □

3.2 Consensus is Impossible without Unique Ids

Having proved that consensus is impossible with crash failures, we consider the conditions under which it remains impossible *without* crash failures. Recall, in wireless networks, unlike wired networks, the network configuration might be ad hoc, preventing nodes from having full *a priori* information on the participants. Accordingly, in this section and the next we explore the network information required to solve consensus. We start here by investigating the importance of unique ids. We call an algorithm that *does not* use unique ids an *anonymous* algorithm. We prove below that consensus is impossible with anonymous algorithms, even if nodes know the network size and diameter. We then provide a corollary that extends this result to the standard asynchronous network model. To the best of our knowledge, this is the first result on the necessity of unique ids for consensus in multihop message passing networks.

Result. To prove our main theorem we leverage an indistinguishability result based on the network topologies shown in Figure 1. Due to the careful construction of these networks, we cannot prove our impossibility holds for all n and D (network B, for example, requires that n be divisible by 3). We can, however, prove that for every sufficiently large (even) D and n, the problem is impossible for D and some $n' = \Theta(n)$.

Theorem 3.3. *There exists a constant integer $c \geq 1$, such that for every even diameter $D \geq 4$ and network size $n \geq D$, there exists an $n' \in \{n, ..., c \cdot n\}$, such that no anonymous algorithm $\mathcal{A}$ guarantees to solve consensus in our abstract MAC layer model in all networks of diameter D and size n'.*

Given some D and n that satisfy the theorem constraints, let k be the smallest integer $k \geq 0$ such that $3(\frac{D-2}{2} + k) + 12 \geq n$. Set $n' = 3(\frac{D-2}{2} + k) + 12$. Consider networks A and B from Figure 1, instantiated with $d = \frac{D-2}{2}$ and k set to the value fixed above in defining n'. In the case of network A, set the clique C to contain enough nodes to bring the total count in that network to equal the number of nodes in B. The following claim follows from the structure of these networks and our definitions of k and d.

Claim 3.4. *Networks A and B, instantiated with the values described above, have size n' and diameter D.*

We define the *synchronous scheduler* in our model to be a message scheduler that delivers messages in lock step rounds. That is, it delivers all nodes' current message to all recipients, then provides all nodes with an ack, and then moves on to the next batch of messages. Furthermore, we assume no global time (or, equivalently, some small amount of time that we can define as needed in the below proof) passes between these *synchronous steps.* Fix some consensus algorithm $\mathcal{A}$ that does not use unique ids. For $b \in \{0, 1\}$, let α_B^b be the execution of $\mathcal{A}$ in network B (see the right network in Figure 1) with all nodes starting with initial value b and message behaviors scheduled by the synchronous scheduler. The following lemma follows directly from the definition of consensus and the fairness of the synchronous scheduler.

Lemma 3.5. *There exists some $t \geq 0$ such that for $b \in \{0, 1\}$, α_B^b terminates by synchronous step t with all nodes deciding b.*

Next, let α_A be an execution of $\mathcal{A}$ in network A (see the left network in Figure 1) defined as follows: (1) all nodes in one gadget start with initial value 0 (call these nodes A_0), all nodes in the other copy of the gadget start with initial value 1 (call these nodes A_1); (2) the bridge node q and the nodes in component C start with arbitrary initial values; and (3) we fix the scheduler to schedule the steps of A_0 and A_1 like the synchronous scheduler for for t steps (for the t fixed in Lemma 3.5), while delaying any message from node q being delivered until after these t steps are complete. After this point, the scheduler can behave as the synchronous scheduler for the full network.

The key argument in our proof is that a node in A_b cannot distinguish itself during the first t steps of α_A from the same node in α_B^b. Intuitively, this follows because the network in B is carefully constructed to be symmetric, so nodes cannot tell if they are communicating with one copy of the network A gadget or multiple copies. To formalize this argument, we introduce some notion that relates network A to B. Notice that network B consists of three copies of the gadget from network A (with some of the edges from the connector node copies swapped to interconnect the copies). For each node u in a network A gadget, therefore, we can define S_u to be the set containing the three nodes in network B that correspond to u: that is, the nodes in u's position in the three gadget copies of B). For example, consider node c in the network A gadget shown in Figure 1. By our above definition, $S_c = \{c_1, c_2, c_3\}$. We can now formalize our indistinguishability.

Lemma 3.6. *Fix some $b \in \{0, 1\}$ and some node u in A_b. The first t steps of u in α_A are indistinguishable from the first t steps of the three nodes in S_u in α_B^b.*

Proof. We begin by noting the following property of our networks that follows directly from its structure (in the following, we use the notation N_{A_b} to indicate the neighbor function of the subgraph of network A consisting only of the nodes in A_b): (*) *Fix any $u \in A_b$ and $u' \in S_u$. For every $v \in N_{A_b}(u)$, u' is connected to exactly one node in S_v. There are no other edges adjacent to u' in B.* We now leverage property (*) in proving the following induction argument, which itself directly implies our lemma statement. The below induction is on the number of synchronous steps in the α executions.

Hypothesis: $\forall u \in A_b$, $0 \leq r \leq t$: after r steps, u in α_A has the same state as the nodes in S_u in α_B^b.

Basis ($r = 0$): Because we assume no unique ids and the same initial values for all relevant nodes, the hypothesis is trivially true after 0 steps.

Step: Assume the hypothesis holds through some step $r, 0 \leq r < t$. We will now show it holds for step $r+1$. By our hypothesis, for each $w \in A_b$, the nodes in S_w will send the same message as w during step $r+1$ (as this message is generated deterministically by the nodes' state after step r). Now consider a particular

$u \in A_b$ and a particular copy $u' \in S_u$ in network B. By property (*), for each node $v \in N_{A_b}(u)$ that sends a message to u in $r+1$, u' is connected to a single node in S_v. By our above argument, this node in S_v will send the same message to u' as v sends to u. Furthermore, (*) establishes that there are no other edges to u' that will deliver messages at this point. It follows that u' will receive the same message set in $r+1$ in α_B^b as u receives in $r+1$ in α_A. They will end $r+1$, therefore, in the same state. □

We now leverage Lemma 3.6 to prove our main theorem.

Proof of Theorem 3.3. Assume for contradiction that there exists an anonymous algorithm $\mathcal{A}$ that guarantees to solve consensus for a diameter D and network size n' specified to be impossible by the theorem statement. Fix some nodes $u \in A_0$ and $v \in A_1$ such that u and v are in the same position in their respective gadgets in network A. Fix some $w \in S_u = S_v$. By Lemma 3.5, w decides 0 within t steps of α_B^0. Combining this observation with Lemma 3.6, applied to u and $b = 0$, it follows that u will decide 0 in α_A. By agreement, it follows that all nodes must decide 0 in α_A—including v. We can, however, apply this same argument to α_B^1, v and $b = 1$, to determine that v decides 1 in α_A. A contradiction. □

We conclude with a corollary for the standard asynchronous model that follows from the fact that our model is strictly stronger and this result concerns a lower bound.

Corollary 3.7. *There exists a constant integer $c \geq 1$, such that for every even diameter $D \geq 4$ and network size $n \geq D$, there exists an $n' \in \{n, ..., c \cdot n\}$, such that no anonymous algorithm $\mathcal{A}$ guarantees to solve consensus in the asynchronous network model with broadcast communication and no advance knowledge of the network topology, in all networks of diameter D and size n'.*

3.3 Consensus is Impossible without Knowledge of n

In Section 3.2, we proved that consensus in our model requires unique ids. Here we prove that even with unique ids and knowledge of D, nodes still need knowledge of n to solve the problem (in multihop networks). Our strategy for proving this theorem is an indistinguishability argument of a similar style to that used in Section 3.2. In more detail, consider network K_D with diameter D shown in Figure 2. Imagine that we start the $D+1$ nodes in sub-graph L_D^1 (resp. L_D^2) with initial value 0 (resp. 1). If we delay message delivery long enough between L_{D-1} and its neighbors in K_D, the nodes in L_D^i cannot distinguish between being partitioned in K_D or executing by themselves in a network. Diameter knowledge does not distinguish these cases.

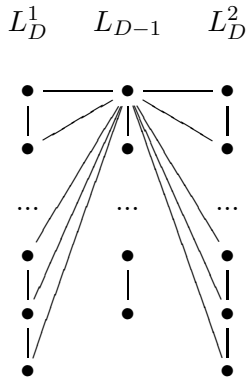

Figure 2: The K_D network. Note: L_{D-1} contains D nodes.

To formalize this argument, we first assume w.l.o.g. that nodes continually send messages. In the following, fix some consensus algorithm $\mathcal{A}$. Let L_d, for integer $d \geq 1$, be the network graph consisting of $d+1$ nodes in a line. Let α_d^b, for $b \in \{0,1\}$ and some integer $d \geq 1$, be the execution of $\mathcal{A}$ in L_d, where all nodes begin with initial value 0 and message behavior is scheduled by the synchronous scheduler (defined in Section 3.2). The following lemma follows from the validity and termination properties of consensus.

Lemma 3.8. *There exists some integer $t \geq 0$, such that for every $b \in \{0,1\}$ and $d \geq 1$, α_d^b terminates after t synchronous steps with all nodes deciding b.*

For a given diameter $D > 1$, we define the network graph K_D to consist of two copies of L_D (call these L_D^1 and L_D^2) and the line L_{D-1}, with an edge added from every node in L_D^1 and L_D^2 to some fixed endpoint of the L_{D-1} line. Notice that by construction, K_D has diameter D. (See Figure 2.) Next, we define the *semi-synchronous scheduler*, in the context of network graph K_D, to be a message scheduler that delivers messages amongst nodes in L_D^1 and amongst nodes in L_D^2, in the same manner as the synchronous scheduler for t

synchronous steps (for the t provided by Lemma 3.8). During this period, the semi-synchronous scheduler does *not* deliver any messages from the endpoint of the L_{D-1} line to nodes in L_D^1 or L_D^2. After this period, it behaves the same as the synchronous scheduler. Let β_D be the execution of $\mathcal{A}$ in K_D with: (1) all nodes in L_D^1 starting with initial value 0; (2) all nodes in L_D^2 starting with initial value 1; (3) all nodes in L_{D-1} starting with arbitrary initial values; and (4) the semi-synchronous scheduler controlling message actions. With these definitions established, we can prove our main theorem.

Theorem 3.9. *For every $D > 1$, no algorithm $\mathcal{A}$ guarantees to solve consensus in our abstract MAC layer model in all networks of diameter D.*

Proof. Assume for contradiction that $\mathcal{A}$ guarantees to solve consensus in all networks of diameter D, for some fixed $D > 1$. By the definition of the semi-synchronous scheduler, it is straightforward to see that β_D is indistinguishable from α_D^0 for nodes in L_D^1, and indistinguishable from α_D^1 for nodes in L_D^2, for the first t synchronous steps. Combining Lemma 3.8 with our indistinguishability, we note that nodes in L_D^1 will decide 0 in β_D while nodes in L_D^2 will decide 1. Therefore, $\mathcal{A}$ does not satisfy agreement in β_D. We constructed K_D, however, so that it has a diameter of D. Therefore, $\mathcal{A}$ guarantees to solve consensus (and thus satisfy agreement) in this network. A contradiction. □

3.4 Consensus Requires $\Omega(D \cdot F_{ack})$ Time

The preceding lower bounds all concerned computability. For the sake of completeness, we conclude by considering complexity. The $\Omega(D \cdot F_{ack})$ time bound claimed below is established by a partitioning argument.

Theorem 3.10. *No algorithm can guarantee to solve consensus in our abstract MAC layer model in less than $\lfloor \frac{D}{2} \rfloor F_{ack}$ time.*

Proof. Fix some D. Consider a line of diameter D consisting of nodes $u_1, u_2, ..., u_{D+1}$, arranged in that order. Consider an execution of a consensus algorithm in this network with a variant of the synchronous scheduler from Section 3.2 that delays the maximum F_{ack} time between each synchronous step. In $\lfloor \frac{D}{2} \rfloor F_{ack}$ time, the endpoints cannot hear from beyond their nearest half of the line. If we assume they must decide by this deadline we can apply a standard partitioning argument to create an agreement violation. In particular, if one half starts with initial value 0 and the other initial value 1, the endpoint of the first half must decide 0 and the endpoint of the second must decide 1 (by indistinguishability and validity), creating an agreement violation. □

4 Upper Bounds

In Section 3, we proved fundamental limits on solving consensus in our model. In this section, we prove these bounds optimal with matching upper bounds. We consider both *single hop* (i.e., the network graph is a clique) and *multihop* (i.e., the network graph is an arbitrary connected graph) networks. Due to the impossibility result from Section 3.1, we assume no crash failures in the following.

4.1 Consensus in Single Hop Networks

Here we describe an algorithm—*two-phase consensus*—that guarantees to solve consensus in single hop network topologies in an optimal $O(F_{ack})$ time. It assumes unique ids but does not require knowledge of n. This opens a separation with the standard broadcast asynchronous model (which does not include acknowledgments) where consensus is known to be impossible under these conditions [1].

Algorithm 1 Two-Phase Consensus (for node u)

```
1: v ← initial value from {0, 1}
2: R_1 ← {⟨phase 1, id_u, v⟩}
3:                                                    ▷ Phase 1
4: broadcast(⟨phase 1, id_u, v⟩)
5: while waiting for ack do
6:     add received messages to R_1
7: end while
8: if ⟨phase 1,*,1 − v⟩ ∈ R_1 or ⟨phase 2,*, bivalent⟩ ∈ R_1 then
9:     status ← bivalent
10: else
11:    status ← decided(v)
12: end if
13:                                                   ▷ Phase 2
14: broadcast(⟨phase 2, id_u, status⟩)
15: R_2 ← {⟨phase 2, id_u, status⟩}
16: while waiting for ack do
17:    add received messages to R_2
18: end while
19: W ← every unique id in R_1 and R_2                ▷ Witness list created
20: while ∃id ∈ W s.t. ⟨phase 2, id,*⟩ ∉ R_1 ∪ R_2 do
21:    add received phase 2 messages to R_2
22: end while
23: if ⟨phase 2, *, decided(0)⟩ ∈ R_2 then
24:    decide 0
25: else
26:    decide 1
27: end if
```

The pseudocode for our algorithm is presented in Algorithm 1. Here we summarize its operation: Each node u executes two *phases*. At the beginning of the first phase, u broadcasts its unique id and its initial value $v_u \in \{0, 1\}$. Node u considers its first phase complete once it receives an acknowledgment for its first broadcast. At this point, u will choose its *status*. If u has seen evidence of a different initial value in the system by this point (i.e., it sees a phase 1 message for a different value or a *bivalent* phase 2 message), it sets its status to *bivalent*. Otherwise, it sets it to *decided* v_u. Node u now begins phase 2 by broadcasting its status and id. Once u finishes this phase 2 broadcast, it has two possibilities. If its status is *decided*, then it can decide its initial value and terminate. Otherwise, it constructs a *witness set* W_u, consisting of every node it has heard from so far in the execution. It waits until it has received a phase 2 message from *every* node in W_u. At this point, if the set contains any message of the form *decided* v_w, then it decides v_w. Otherwise, it decides default value 1. We now establish the correctness of this strategy.

Theorem 4.1. *The two-phase consensus algorithm solves consensus in $O(F_{ack})$ time in our abstract MAC layer model in single hop networks with unique ids.*

Proof. Validity and termination are straightforward to establish. We turn our attention, therefore, to agreement. If no node ends up with *status* = *decided*(0), then 1 is the only possible decision value. The interesting case, therefore, is when some node u does set *status* $\leftarrow$ *decided*(0) after its phase 1 broadcast completes. Let S be the subset of nodes that began with initial value 1 (if any). By assumption, u's phase 1 broadcast completed before any node in S, as, otherwise, u would have seen evidence of a 1 before setting $status$, preventing it from choosing *decided*(0). It follows that every node in S must set $status$ to *bivalent.* We know, therefore, that it is impossible to have both *decided*(1) and *decided*(0) in the system. We are left to show that if there is *decided*(0) in the system, then all nodes end up deciding 0. As before, let u be a node with status *decided*(0). Now let v be a node with status *bivalent*. We consider two cases concerning u and v's interaction.

In the first case, assume v receives a message from u before v finishes its phase 2 broadcast. It follows that u will be placed in v's witness list, W. The algorithm now requires v to wait for u's phase 2 broadcast before deciding. It will therefore see that u has a status of *decided*(0), requiring v to decide 0. In the second case, v does not receive a message from u before v finishes its phase 2 broadcast. Accordingly, u is not in v's witness set W. This might be problematic as it could allow v to decide before it sees a *decided*(0) message. Fortunately, we can show this second case cannot happen. If v had not heard *any* message from u by the time it finished its phase 2 broadcast, it follows that u receives this broadcast before it finishes its phase 1 broadcast. But v's phase 2 broadcast has a *bivalent* status. By the algorithm, this would prevent u from setting its status to *decided*(0)—contradicting our assumption that u has a *decided* status. □

4.2 Consensus in Multihop Networks

We now describe a consensus algorithm for the multihop setting that guarantees to solve consensus in $O(D \cdot F_{ack})$ time. It assumes unique ids and knowledge of n (as required by the lower bounds of Section 3), but

makes no additional assumptions about the participants or network topology. Notice, this solution does not *replace* the single hop algorithm of Section 4.1, as this previous algorithm: (1) is simpler; (2) has a small constant in its time complexity (i.e., 2); and (3) does not require knowledge of n.[5]

Our strategy for solving consensus in this setting is to leverage the logic of the PAXOS consensus algorithm [30, 31]. This algorithm was designed and analyzed for the asynchronous network model with bounded crash failures. Here we apply the logic to our wireless model with no crash failures. The main difficulty we face in this effort is that nodes do not know the topology of the network or the identity of the other participants in advance. To overcome these issue we connect the PAXOS logic with a collection of sub-routines we call *services*, which are responsible for efficiently delivering messages, electing the leaders needed by PAXOS for liveness, and telling the proposers when to generate new proposal numbers. We call this combination of PAXOS logic with our model-specific services *wireless PAXOS* (wPAXOS).

If we were satisfied with a non-optimal $O(n \cdot F_{ack})$ time complexity, the communication services could be implemented with a simple flooding logic (the first time you see a message, re-broadcast), and the leader election service could simply return the largest id seen so far. To obtain an optimal $O(D \cdot F_{ack})$ time complexity, however, requires a more intricate solution. In particular, when a proposer is waiting to hear from a majority of acceptors, we cannot afford for it to receive each response individually (as each message can only hold a constant number of unique ids, and this would therefore require a proposer to receive $\Theta(n)$ messages). Our solution is to instead have nodes execute a distributed Bellman-Ford style iterative refinement strategy to establish shortest-path routing trees rooted at potential leaders. We design this service such that once the leader election service stabilizes, a tree rooted at this leader will complete soon after (if it is not already completed). These trees are then used to safely *aggregate* responses from acceptors: a strategy that leverages the fact that PAXOS only requires the total *count* of a given response type, not the individual responses.[6] This aggregation strategy reduces the time to gather responses (after stabilization) from $O(n \cdot F_{ack})$ to $O(D \cdot F_{ack})$.

The final optimization needed to adapt PAXOS to our model is the change service. We need the eventual leader to generate proposals *after* the leader election and tree services stabilize, so it can reap the benefits of efficient acceptor response aggregation. At the same time, however, the leader cannot generate *too many* new proposals after this point, or each new proposal may delay the previous. The key property of our change service is that it guarantees that the leader will generate $\Theta(1)$ new proposal after stabilization (assuming there is no decision yet in the network).

4.2.1 Algorithm

Our algorithmic strategy is to implement the logic of the classic PAXOS asynchronous agreement algorithm [30, 31] in our abstract MAC layer model. To do so, we implement and analyze a collection *services* that can be connected to the high-level PAXOS logic to run it in our model. These services are responsible for disseminating proposer messages and acceptor responses (in an efficient manner), as well as notifying the high level PAXOS logic when to start over with a new proposal number. They also provide the leader election service needed for liveness. As mentioned, we call this combination of the high-level PAXOS logic with our model-specific support services, *wireless PAXOS* (wPAXOS).

We note, that if we did not care about time complexity, our services could disseminate messages and responses with simple flooding services. To achieve an optimal $O(D \cdot F_{ack})$ time, however, requires a more complicated strategy. In more detail, our services build, in a distributed fashion, an eventually stabilized shortest path tree rooted at the eventual leader in the network. Once stabilized, acceptors can efficiently send their responses to the leader by aggregating common response types as they are routed up the tree.

We proceed below by first describing these support services, and then describing how to connect them to the standard PAXOS logic. We conclude by proving our needed safety and liveness properties for the

[5]This lack of knowledge of n does not violate the lower bound of Section 3.3, as this lower bound requires the use of a multihop network topology.

[6]A technicality here is that these responses sometimes include prior proposals; we handle this issue by simply maintaining in aggregated responses the prior proposal—if any—with the largest proposal number of those being aggregated.

Algorithm 2 Leader Election Service (for node u)

```
1:  procedure ON INITIALIZATION
2:      Ω_u ← id_u
3:      UpdateQ(⟨leader, id_u⟩)
4:  end procedure

5:  procedure RECEIVE(⟨leader, id⟩)
6:      if id > Ω_u then
7:          Ω_u ← id
8:          UpdateQ(⟨leader, id⟩)
9:      end if
10: end procedure

11: procedure UPDATEQ(⟨leader, id⟩)
12:     empty leader queue and enqueue ⟨leader, id⟩
13: end procedure
```

Algorithm 3 Change Service (for node u)

```
1:  procedure ON INITIALIZATION
2:      lastChange ← −∞
3:  end procedure

4:  procedure ONCHANGE                    ▷ Ω_u or dist_u updated.
5:      lastChange ← time_stamp()
6:      UpdateQ(⟨change, lastChange, id_u⟩)
7:  end procedure

8:  procedure RECEIVE(⟨change, t, id⟩)
9:      if t > lastChange then
10:         lastChange ← t
11:         UpdateQ(⟨change, t, id⟩)
12:     end if
13: end procedure

14: procedure UPDATEQ(⟨change, t, id⟩)
15:     empty the change queue then enqueue ⟨change, t, id⟩
16:     if Ω_u = id_u then
17:         GenerateNewPAXOSProposal()
18:     end if
19: end procedure
```

Algorithm 4 Tree Building Service (for node u)

```
1:  procedure ON INITIALIZATION
2:      ∀v ≠ u : dist[id_v] ← ∞ and parent[id_v] ← ⊥
3:      dist[id_u] ← 0 and parent[id_u] ← id_u
4:      UpdateQ(⟨search, id_u, 1⟩)
5:  end procedure

6:  procedure RECEIVE(m = ⟨search, id, h⟩)
7:      if h < dist[id] then
8:          dist[id] ← h
9:          parent[id] ← m.sender
10:         UpdateQ(⟨search, id, h + 1⟩)
11:     end if
12: end procedure

13: procedure UPDATEQ(⟨search, id, h⟩)
14:     enqueue ⟨search, id, h⟩ on tree queue
15:     discard any message for id with hop count h′ > h
16:     move message (if any) with id Ω_u to front of tree queue
17: end procedure

18: procedure ONLEADERCHANGE      ▷ Called when Ω_u changes
19:     move message (if any) with id Ω_u to front of tree queue
20: end procedure
```

Algorithm 5 Broadcast Service (for node u)

```
1:  while true do
2:      wait for at least one queue from
3:          {tree, leader, change} to become non-empty
4:      dequeue a message from each non-empty queue and
5:          combine into one message m.
6:      broadcast(m) then wait for ack
7:  end while
```

Figure 3: Support services used by wPAXOS. Notice, the broadcast service schedules message broadcasts from the queues maintained by the other three services. We assume that when a message is received it is deconstructed into its constituent service messages which are then passed to the `receive` procedure of the relevant service. This basic receive logic is omitted in the above.

resulting combined algorithm. In the following, we assume the reader is already familiar with the PAXOS algorithm (if not, see [31]), and will focus on the new service algorithms specific to our model it uses and how it uses them.

Services. Our wPAXOS algorithm requires the four support services (see Figure 3 for the pseudocode). The first three, *leader election*, *change*, and *tree building* each maintain a message queue. The fourth, *broadcast*, is a straightforward loop that takes messages from the front of these queues, combines them, then broadcasts the combined message—allowing the algorithm to multiplex multiple service on the same channel. In the following, therefore, we talk about the messages of the leader election, change, and tree building services as if they were using a dedicated channel.

Leader Election. This service maintains a local variable Ω_u, containing an id of some node in the network, at each node u. The goal of this service is to eventually stabilize these variables to the same id network-wide. It is implemented with standard flooding logic.

Tree Building. This service attempts to maintain in the network, for each node v, a shortest-path tree rooted at v. The protocol runs a Bellman-Ford style iterative refinement procedure for establishing these trees, with the important optimization that the messages corresponding to the current leader get priority in each node's tree service queue. This optimization ensures that soon after the network stabilizes to a single leader a tree rooted at that leader is completed.

Change. This service is responsible for notifying PAXOS proposers when they should start a new proposal number. The goal for this service is to ensure that the eventual leader u_ℓ starts a new proposal after the other two services have stabilized: that is, after the leader election service has stabilized to u_ℓ across the network, and the tree rooted at u_ℓ is complete. Proposals generated after this point, we will show, are efficiently processed. The service must also guarantee, however, not to generate *too many* changes after this point, as each new proposal can delay termination.

Connecting PAXOS Logic to Support Services. We now describe how to combine the standard PAXOS logic with our model-specific support services to form the wPAXOS algorithm. Recall, in PAXOS there are two roles: *proposers* and *acceptors*.[7] In wPAXOS all nodes play both roles. We describe below how both roles interact with our services.

Proposers. Proposers generate *prepare* and *propose* messages (the latter sometimes called *accept* messages), associated with a given proposal number. A proposal number is a tag (initially 0) and the node's id. The pairs are compared lexicographically. A key detail in any PAXOS deployment is the conditions under which a proposer chooses a new proposal number and starts over the proposal process. In wPAXOS, a proposer starts over when its change service locally calls *GenerateNewPAXOSProposal*(). At this point it increases its tag to be 1 larger than the largest tag it has previously seen or used. If this new proposal number is rejected at either stage, *and* the proposer learns of a larger proposal number in the system during this process (i.e., because an acceptor appended a previous message to its rejection), *and* it is still the leader according to its leader election service, it increases its tag number and attempts a new proposal. If this new proposal also fails, it waits for the change service before trying again. In other words, a proposer will only try up to 2 proposal numbers for each time it is notified by the change service to generate a new proposal.

To disseminate the *prepare* and *propose* messages generated by proposers we assume a simple flooding algorithm (due to its simplicity, we do not show this pseudocode in Figure 3): if you see a proposer message from u for the first time, add it to your queue of proposer messages to rebroadcast. To prevent old proposers messages from delaying new proposer messages we have each proposer maintain the following invariant regarding the message queues used in this flooding: At all times, the queue is updated to ensure: (1) only contains messages from the current leader; and (2) only contains messages associated with the largest proposal number seen so far from that leader.

[7] There is sometimes a third role considered, called *learners*, that are responsible for determining when a decision has been made. As is common, however, we collapse this role with the role of the proposer. When a proposer determines it can can decide a value—i.e., because a proposal for this value was accepted by a majority of acceptors—it does so and floods this decision to the rest of the network.

Acceptors. We now consider the acceptors. When acceptor v generates a response to a *prepare* or *propose* message from proposer u, v labels the response with the destination $parent[\text{id}_u]$ then adds it to its acceptor broadcast queue. This message is then treated like a unicast message: even though it will be broadcast (as this is the only communication primitive provided by our model), it will be ignored by any node except $parent[\text{id}_u]$. Of course, if $v = u$ then the message can skip the queue and simply be passed over to the proposer logic.

To gain efficiency, we have acceptors aggregate messages in their acceptor queue when possible. In more detail, if at any point an acceptor v has in its queue multiple responses of the same type (positive or negative) to the same proposer message, to be sent to same $parent$: it can combine them into a single *aggregated* message. This single message retains the the type of responses as well as the proposal number the responses reference, but replaces the individual responses with a count. We can combine aggregated messages with other aggregated messages and/or non-aggregated messages in the same way (i.e., given two aggregated messages with counts k_1 and k_2, respectively, we can combine them into an aggregated message with count $k_1 + k_2$).

Notice, PAXOS sometimes has acceptors respond to a *prepare* message with a previous proposal (i.e., combination of a proposal number and value). When aggregating multiple messages of this type (i.e., containing previous proposal) we keep only the previous proposal with the largest proposal number. We also assume PAXOS implements the standard optimization that has acceptors, when rejecting a message, append the larger proposal number to which they are currently committed. We aggregate these previous proposals in the same way we do with positive responses to prepare messages—by maintaining only the largest among those in the messages we are aggregating.

Finally, as with the proposers, the acceptors keep their message queue up to date by maintaining the invariant that at all times, the queue is updated to ensure: (1) only contains messages in response to propositions from the current leader; and (2) only contains responses associated with the largest proposal number seen so far from that leader.

Deciding. When a proposer learns it can decide a value val (i.e., because it has learned that at least a majority of acceptors replied with *accept* to a proposal containing val), it will decide val then flood a *decide*(val) message. On receiving this message, a node will decide val.

4.2.2 Analysis

We now prove that our wPAXOS algorithm solves consensus in $O(D \cdot F_{ack})$ time—matching the relevant lower bounds. In the following, let a *message step* (often abbreviated below as just "step") be an event in an execution where a message or ack is received. Notice, because we consider deterministic algorithms and assume that local computation does not require any time, the behavior of an algorithm in our model is entirely describe by a sequence of message steps. Let a *proposition* be a combination of a proposer, a proposer message type (i.e., either *prepare* or *propose*), and a proposal number. For a fixed execution of wPAXOS: let $a(p)$, for a given proposition p, be the number acceptors that generate an affirmative response to proposition p; and let $c(p)$ be the total count of affirmative responses for proposition p received by the originator of p. Similarly, let $c(p, s)$, for step s, be the total count of affirmative responses to p received by the end of step s.

Safety. In the standard asynchronous network model, PAXOS takes for granted that proposers properly count the relevant responses to their propositions, as the model reliably delivers each response, labeled with the acceptor that generated it. In wPAXOS, however, we aggregate responses using shortest-path trees that might change during throughout the execution. Before we can leverage the standard safety arguments for PAXOS, we must first show that this aggregation in dynamic trees does not compromise the integrity of the proposer response counts.

Lemma 4.2. *Fix some execution of wPAXOS. Let p be a proposition generated by u in this execution. It follows that $c(p) \leq a(p)$.*

Proof. In this proof, for a fixed execution, proposition p, node v, and step s, we define $q(p, v, s)$ be the sum of the following values: (1) the total count of affirmative responses to proposition p in node v's acceptor

message queue after step s; (2) the total count of the affirmative responses to p in the message v is in the process of sending to some w after step s, assuming that w has not yet received the message at this point; (3) 1, if the acceptor at v will generate an affirmative response to p at some later step $s' > s$ in the execution. Similarly, let $Q(p, s) = \sum_{v \in V} q(p, v, s)$. Fix some proposition p with originator u. To prove our lemma it is sufficient to prove the following invariant:

(*) For every step $s \geq 0$: $Q(p, s) + c(p, s) \leq a(p)$.

We prove (*) by induction on s. The *basis* ($s = 0$) of this induction is trivial: by definition $Q(p, 0) = a(p)$ and $c(p, 0) = 0$. For the *inductive step* we assume (*) holds through step s. There are three cases for step $s + 1$ that are relevant to our hypothesis.

Case 1*:* Assume $s + 1$ has some acceptor $v \neq u$ receive a message that causes it to generate a response to p. This action transfers quantity 1 from $q(p, v, s)$ (contributed from element (3) of the definition of q) and transfers it either to the messages in v's queue, or, if it is not in the process of sending a message when $s + 1$ occurs, into a message being sent by v. In either case, this value of 1 now shows up in $q(p, v, s + 1)$ either from element (1) *or* (2) of its definition/ It follows that $Q(p, s + 1) = Q(p, s)$ and $c(p, s + 1) = c(p, s)$.

Case 2*:* Assume $s + 1$ has some acceptor $w \neq u$ receive a message m from v that contains affirmative responses to p *and* v addressed the message to w. In this case, the count of affirmative responses contained in this message was moved unchanged from $q(p, v, s)$ to $q(p, w, s + 1)$. Once again $Q(p, s) = Q(p, s + 1)$ and $c(p, s + 1) = c(p, s)$.

Case 3*:* Assume step s has u receive a message with k affirmative responses to p that is addressed to u. In this case, u adds k to its count and discards the message. It follows that $Q(p, s + 1) = Q(p, s) - k$ and $c(p, s + 1) = c(p, s) + k$. □

The below lemma comes from [31], where Lamport shows that proving this lemma provides agreement. If we can prove this below lemma, in other words, then we can apply the arguments of [31] to establish agreement. Notice, this proof leverages Lemma 4.2.

Lemma 4.3. *Fix an execution of wPAXOS. Assume proposer u generates a proposal with value val and proposal number x. It follows that there exists a set S consisting of a majority of acceptors such that either: (a) no acceptor in S has accepted a proposal with number smaller than x, or (b) val is the value of the highest-numbered proposal among all proposals with numbers less than x accepted by the acceptors in S.*

Proof. Fix some execution α. Let p' be a proposal proposition generated by u for proposal number x. Let p be the prepare proposition from u with this same number x that must, by the definition of the algorithm, precede p'. Let A be the set of acceptors that commits to p. Let s be the step at which $c(p, s)$ becomes greater than $n/2$ for the first time. We define a directed graph $G(\alpha, p, s) = (V, E')$ as follows. Add (w, v) to E' iff at some step $s' \leq s$ in α, acceptor w sent a message about a positive response to p to v. Let $A_u \subset A$ be the subset of A that have a path to u in $G(\alpha, p, s)$.

Consider the case where at least one node in A_u had previously committed to a proposal number $\hat{x}$ when p arrived. We know $\hat{x} < x$ as this node subsequently committed to x. Let $x_{max} \geq \hat{x}$ be the largest such proposal number of a previous commitment and val_{max} be the corresponding value. Let P be the set of previous proposals that u receives in response to p. We argue that (x_{max}, val_{max}) is in P and has the largest proposal number of any proposal in P. If this is true, it follows that u will adopt val_{max} for its proposal p', as needed by the lemma.

To show this is true, let v_{max} be an acceptor in A_u that previously committed to that proposal. By the definition of $G(\alpha, p, s)$, there is a casual path of messages on which this proposal could travel to u. The only way it could be discarded before arriving at u is if at some point a larger proposal in response to p joins the path from v_{max} to u, *before* the x_{max} proposal has been sent forward. By assumption, however, no process in A_u has a larger proposal number. It follows that this larger proposal must have originated from some $v' \notin A_u$. However, if this previous proposal can intercept the path from v_{max} to u before v_{max}'s messages have passed that point, then, by the definition of $G(\alpha, p, s)$. u must have a path to u in $G(\alpha, p, s)$, and therefore must be in A_u.

We have now shown that properties (a) and (b) hold's for A_u. To prove the lemma, we are left to show that A_u must hold a majority of acceptors. Assume for the sake of contradiction that $|A_u| \leq n/2$. At step s, however, proposer u has $c(p, s) > n/2$. By the definition of $G(\alpha, p, s)$, there is no causal path of messages being sent and received from nodes in $V \setminus A_u$ to u that arrive at u before it counts a majority of acceptances. Therefore, the execution α' in which *no* node in $V \setminus A_u$ commits to p is indistinguishable from α with respect to u through step s. In α', however, $|A| \leq n/2$ and therefore $a(p) \leq n/2$. In this same execution, we also know $c(p) > n/2$. The result: in α', $c(p) > a(p)$. This contradicts Lemma 4.2 □

Part of the difficulty tackled by wPAXOS is efficiently disseminating responses form acceptors even though the messages are of bounded size (i.e., can only hold $O(1)$ ids). If message size was instead unbounded, we could simply flood every response we have seen so far. Given this restriction, however, we must make sure that the proposal numbers used by proposers in wPAXOS do not grow too large. In particular, we show below that these proposal numbers always remain small enough to be represented by $O(\log n)$ bits (messages must at least this large by the assumption that they can hold a constant number of ids in a network of size n).

Lemma 4.4. *There exists a constant k such that the tags used in proposal numbers of wPAXOS are bounded by $O(n^k)$.*

Proof. Each node u can only locally observe $O(n^2)$ change events. its leader variable can only have n possible values and these values strictly increase. And for each v, $dist[v]$ at u can only have n values it strictly decreases. Each change event can lead to at most 2 new proposal numbers at each node in the network (recall: the algorithm restricts a proposer to attempting at most 2 new numbers in response to a given call of *GenerateNewPAXOSProposal*). We also note that there are n total nodes and the tags in proposal numbers are increased to be only 1 larger than the largest tag previously seen. A straightforward induction argument bounds the largest possible tag size as polynomial in n as needed. □

Liveness. We now establish that all nodes in wPAXOS will decide by time $O(D \cdot F_{ack})$. The main idea in this proof is to consider the behavior of the algorithm after the tree and leader election service stabilize. WE show this stabilization occurs in $O(D \cdot F_{ack})$ time and after this point the leader will continue to generate proposals and end up reaching a decision within an additional $O(D \cdot F_{ack})$ time.

Lemma 4.5. *Fix some execution of wPAXOS. Every node decides in $O(D \cdot F_{ack})$ time.*

Proof. Let $\hat{s}$ be the step that generates the *final* change in this execution. Let $t(\hat{s})$ be the global time at which this step occurs. Repurposing a term from the study of partially synchronous model, we call $t(\hat{s})$ the *global stabilization time* (GST) as it defines a time by which point: (a) the whole network has stabilized to the same leader, ℓ; (b) the tree defined by $parent[\ell]$ pointers in the network has stabilized to a shortest-path spanning tree rooted at ℓ.

Consider ℓ's behavior after the GST. We know that $\hat{s}$ generates a change message. Because this change message has a timestamp as larger (or larger) than other change message in the execution, every node will receive a change message with this timestamp for the first time somewhere in the $[t(\hat{s}), t(\hat{s}) + O(D \cdot F_{ack})]$. This is the last change message any node will pass on to their *UpdateQ* procedure in the change service. This is significant, because it tells us that ℓ will generate a new proposition at some on or after the GST, but not *too much* after. And, this is the last time the change service will cause ℓ to generate a proposition.

We now bound the efficiency of ℓ's propositions after the GST. Notice, after GST only messages from ℓ can be stored in proposer queues. When ℓ generates a new proposition message with a larger proposal number than any previous such messages, it will necessarily propagate quickly—reaching a major it of acceptors in $O(D \cdot F_{ack})$ time. We can show that acceptor responses return to ℓ with the same bound, as every such response is at most distance D from ℓ in the shortest path tree, and can be delayed by at most $O(F_{ack})$ time at each hop (if multiple responses arrive at the same node they will simply be aggregated into one response)

Now we consider the fate of ℓ's propositions after GST. We know from above that it generates a new proposition with a new proposal number on or after GST. If receives enough commits to its prepare message

for this proposal number, then it follows it will go on to gather enough accepts to decide (as no other proposer, after GST, can have its proposals considered). If it fails to gather enough commits, it will move on to the next proposal after counting a majority of the acceptors rejecting its prepare. By the algorithm, however, it will learn the largest proposal number that one of this set has previously committed to. Its subsequent proposal number will be larger, and therefore this same majority will end up sending him enough commits to move on toward decision. It follows that a constant number of propositions is sufficient for ℓ to decide.

To tie up the final loose ends, we note that once ℓ decides, all nodes decide within an additional $O(D \cdot F_{ack})$ time, and that GST is bounded by $O(DF_{ack})$ as well: it takes this long for the leader election service to stabilize, after which the tree rooted at the leader must be stabilized within in addition $O(D \cdot F_{ack})$ time. □

Pulling Together the Pieces. We are now ready to combine our key lemmas to prove the final correctness of wPAXOS.

Theorem 4.6. *The wPAXOS algorithm solves consensus in $O(D \cdot F_{ack})$ time in our abstract MAC layer model in any connected network topology, where D is the network diameter and nodes have unique ids and knowledge of network size.*

Proof. The *validity* property is trivial. The *termination* property as well as the $O(D \cdot F_{ack})$ bound on termination follow directly from Lemma 4.5. To satisfy agreement we combine Lemma 4.3 with the standard argument of [31]. □

5 Conclusion

Consensus is a key primitive in designing reliable distributed systems. Motivated by this reality and the increasing interest in wireless distributed systems, in this paper we studied the consensus problem in a wireless setting. In particular, we proved new upper and lower bounds for consensus in an abstract MAC layer model—decreasing the gap between our results and real deployment. We prove that (deterministic) consensus is impossible with crash failures, and that without crash failures, it requires unique ids and knowledge of the network size. We also establish a lower bound on the time complexity of any consensus solution. We then present two new consensus algorithms—one optimized for single hop networks and one that works in general multihop (connected) networks.

In terms of future work, there are three clear next steps that would help advance this research direction. The first is to consider consensus in an abstract MAC layer model that includes unreliable links in addition to reliable links. The second is to consider what additional formalisms might allow deterministic consensus solutions to circumvent the impossibility concerning crash failures. In the classical distributed systems setting, failure detectors were used for this purpose. In the wireless world, where, among other things, we do not always assume *a priori* knowledge of the participants, this might not be the most natural formalism to deploy. The third direction is to consider randomized algorithms, which might provide better performance and the possibility of circumventing our crash failure, unique id, and/or network size knowledge lower bounds.

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

[6] 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 International Symposium on Principles of Distributed Computing*, 1987.

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

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

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

[10] Gregory Chockler, Murat Demirbas, Seth Gilbert, Calvin Newport, and Tina Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proceedings of the International Symposium on Principles of Distributed Computing*, 2005.

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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 Symposium on Distributed Computing*, 2013.

[24] 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.

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

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

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

[28] Fabian Kuhn, Nancy Lynch, and Calvin Newport. The Abstract MAC Layer. In *Proceedings of the International Symposium on Distributed Computing*, 2009.

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

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

[31] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[32] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[33] 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.

[34] Thomas Moscibroda and Roger Wattenhofer. Maximal Independent Sets in Radio Networks. In *Proceedings of the International Symposium on Principles of Distributed Computing*, 2005.

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

[36] Achour Mostefaoui and Michel Raynal. Solving consensus using chandra-touegs unreliable failure detectors: A general quorum-based approach. *Lecture Notes in Computer Science*, 1693:49–63, 1999.

[37] D. Peleg. Time-Efficient Broadcasting in Radio Networks: a Review. In *Proceedings of the Conference on Distributed Computing and Internet Technology*, 2007.

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

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

[40] 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.

[41] 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.