Asynchronous Leader Election and MIS Using Abstract MAC Layer

Nancy Lynch
CSAIL, MIT
Cambridge, MA - 02139, USA
lynch@csail.mit.edu

Tsvetomira Radeva
CSAIL, MIT
Cambridge, MA - 02139, USA
radeva@csail.mit.edu

Srikanth Sastry
CSAIL, MIT
Cambridge, MA - 02139, USA
sastry@csail.mit.edu

ABSTRACT

We study leader election (LE) and computation of a maximal independent set (MIS) in wireless ad-hoc networks. We use the abstract MAC layer proposed in [14] to divorce the algorithmic complexity of solving these problems from the low-level issues of contention and collisions. We demonstrate the advantages of such a MAC layer by presenting simple asynchronous deterministic algorithms to solve LE and MIS and proving their correctness. First, we present an LE algorithm for static single-hop networks in which each process sends no more than three messages to its neighbors in the system. Next, we present an algorithm to compute an MIS in a static multi-hop network in which each process sends a constant number of messages to each of its neighbors in the communication graph.

1. INTRODUCTION

The popularity and emerging ubiquity of wireless networks has attracted significant attention to solving classic distributed problems on ad-hoc wireless systems. The broadcast nature of wireless networks introduces new challenges for algorithm design. Specifically, the contention for exclusive access to the broadcast channel among multiple processes plays a significant role in the design and correctness of algorithms for wireless networks. Consequently, algorithms in wireless networks have to first resolve contention before proceeding to problem-specific communication of information that ultimately solves the problem at hand.

In fact, many algorithms proposed for wireless networks conflate the challenges associated with low-level network properties such as collisions and contention with the challenges associated with solving the actual problem. Consequently, algorithms designed for a given wireless model may not behave correctly in a slightly different model. Additionally, algorithm designers are required to grapple with low-level problems such as contention management repeatedly, making it difficult to highlight interesting high-level algorithmic issues.

Recently, in [14, 15, 16], the following solution was proposed to address the above concerns. Posit the existence of an *abstract MAC layer* that can be implemented on top of wireless ad hoc networks such that the MAC layer addresses the issues of collisions and contention exclusively. Thus, the processes using this abstract MAC layer can send and receive messages, and the MAC layer reliably delivers these messages to all the processes within the communication radius of the transmitting process.

More specifically, the abstract MAC layer [14, 15, 16] delivers messages reliably within the local neighborhood and provides an acknowledgment of the message delivery to the sender. Furthermore, the abstract MAC layer may provide guarantees on the time it takes to deliver these messages. However, the abstract MAC layer does not provide a sender with information about specific recipients of its message. This approach is intended to divide the design and analysis of distributed algorithms (in wireless networks) into two parts. The first part is the abstract MAC layer itself, which addresses the issues of contention and collisions over the physical network. The second part uses the abstract MAC layer to solve the problem at hand while being agnostic to the issues of low-level contention.

We illustrate the complexity of solving problems directly on the physical layer and the advantages offered by the MAC layer by focusing on two well-known problems: leader election (LE) and maximal independent set (MIS). Briefly, in leader election, all the processes within the network elect a single unique process to be the 'leader'; in maximal independent set, the system selects a subset S of processes such that each process in the system is either in S or a neighbor of a process in S, but not both.

Summary of results This paper focuses on demonstrating the advantages offered by the abstract MAC layer in designing high-level algorithms in wireless networks. First, we show that leader election in static single-hop networks using the abstract MAC layer can be solved with an algorithm in which each process sends no more than three messages to its neighbors in the system. Next, we propose another algorithm that computes an MIS in static multi-hop wireless networks using the abstract MAC layer. In this algorithm, each process sends a constant number of messages to each of its neighbors in the communication graph. We provide formal proofs of correctness for both the algorithms and state their time and message complexity.

Organization In Section 2, we discuss current work related to leader election, MIS, and abstract MAC layer in wireless networks. In Section 3, we provide a detailed description of our system model, including the abstract MAC layer. In Section 4, we propose a simple algorithm to solve leader election in single-hop networks using the abstract MAC layer. Similarly, in Section 5, we propose a simple algorithm to compute an MIS in multi-hop networks using the abstract MAC layer. Finally, we end with discussion in Section 6.

2. RELATED WORK

This section provides an overview of algorithms for solving leader election in single-hop wireless networks, computing an MIS in multi-hop wireless networks, and the abstract MAC layer implementation and use.

Leader Election The leader election (LE) problem has been studied in the context of wired networks, shared memory (cf. [1, 18]), and wireless networks (cf. [3, 8, 12, 20, 22, 25]). Here, we focus on LE in single-hop wireless systems.

Several randomized algorithms in [3, 4, 13, 20, 21, 23, 27] use various techniques for solving LE. However, in the aforementioned articles, the overall approach to solving LE can be separated into two steps. The first step estimates the number of processes within the system, and the second step uses this estimate to ensure that exactly one process transmits with high probability; the first process to transmit in isolation is elected the leader. While the mechanism to estimate the number of processes is different for different system models, solving leader election itself is fairly straightforward in the second step. Given an estimate n on the number of processes, each process transmits its ID with probability $1/n$; with high probability, we are guaranteed that some process will transmit in isolation within a constant number of rounds.

Similarly, despite the deterministic algorithms [6, 9, 13] being different for different models of wireless networks, the overall approach for solving LE is the same. These algorithms partition the set of processes into smaller subsets based on some asymmetry (such as process ID) and elect leaders among these smaller subsets. The set of leaders thus obtained are further partitioned into smaller subsets and the procedure is repeated until only one leader remains.

Maximal Independent Set Computing a maximal independent set (MIS) in ad hoc wireless networks is of particular interest because an MIS serves as a *de facto* infrastructure for deploying complex communication services like routing, virtual network backbone, and global broadcast.

Recent investigations into computing an MIS in wireless networks include [17, 19, 24, 25] which employ different techniques, both deterministic and probabilistic, that share common properties. Each process repeatedly participates in a competition with the neighboring processes. The processes that win a competition join the MIS and the processes that lose the competition do not join the MIS. If neighboring processes neither win nor lose a competition, then they reset to a predetermined state and restart the competition. While the techniques appear to be similar to each other, the specific algorithms in the aforementioned articles differ significantly because the low-level network properties vary among the different system models.

Although the various techniques that solve LE and MIS, respectively, share a similar structure, each individual technique exploits network-specific properties. Therefore, if such network-specific properties are abstracted into a system service with a fixed interface, then solving LE or MIS using such an interface is arguably simpler. The construction of this interface itself may be system-dependent, but the algorithms using this interface may be network agnostic; in effect, the algorithms designed for such an interface are portable across systems on which such interfaces are built. In this paper, we argue that the Abstract MAC layer is one such interface, and we demonstrate how the Abstract MAC layer simplifies the algorithms for solving LE and MIS.

Abstract MAC Layer Current work on abstract MAC layer can be classified as either implementing abstract MAC layers on physical networks or designing algorithms that use an abstract MAC layer. In [11], abstract MAC layers are implemented on wireless networks that use analog network coding such as Zigzag decoding [7], and in [10], deterministic and probabilistic abstract MAC layers are implemented in the collision-prone radio network model [2]. Additionally, the results in [10] provide efficient solutions for single-message and multi-message broadcast problems using the abstract MAC layer. The abstract MAC layer is used in [5] to build the *dynamic-graph model* from [26]. In brief, the dynamic-graph model allows communications channels to go up and down, subject to certain restrictions, and thus, constantly change the topology of the network.

3. SYSTEM MODEL

Our model considers a finite set of n processes with unique IDs from a fixed finite ID space. The ID of each process i is denoted id_i and is immutable. Each process in an I/O automaton [18] with a fixed initial state, knows its own ID, but has no knowledge of other processes in the system. Processes communicate with each other through the abstract MAC layer as described next. We assume that processes take steps asynchronously and may either wake up spontaneously, or wake up upon receiving a message from another process.

Notation. We denote an event e that occurs at process i as e_i. Analogously, a variable v at a process i is denoted v_i.

3.1 Abstract MAC Layer

Informally, the abstract MAC layer [14, 15, 16] may be viewed as a reliable local-broadcast service that provides feedback to the sender in the form of an acknowledgment after the message has been successfully delivered. However, the service need not provide information about the particular recipients of the message.

More precisely, the abstract MAC layer is an I/O automaton [18] that provides an interface with $bcast(m)_i$ and $abort(m)_i$ inputs and $rcv(m)_i$ and $ack(m)_i$ outputs for every message m in some fixed alphabet and every process automaton i in the system. The abstract MAC layer is assumed to be implemented on top of the physical wireless network. Next, we describe the behavior of the abstract MAC layer when composed with the physical network and its interaction with processes in the system. Intuitively, $bcast(m)_i$ denotes the broadcast of a message m by process i, $abort(m)_i$ denotes process i aborting an ongoing broadcast of message m, $rcv(m)_i$ denotes the receipt of a message m by process i, and $ack(m)_i$ denotes the receipt of the ack for a previous broadcast of message m by i.

To describe meaningful behaviors of the MAC layer, we assume the following well-formedness conditions for the process automata interacting with the abstract MAC layer when composed with the physical network.

Fix an execution α of the system composed of the physical network, abstract MAC layer, and process automata. We say that α is well-formed iff the following hold: (1) every $abort(m)_i$ is preceded by a $bcast(m)_i$ with no intervening $bcast()_i$, $ack()_i$, or $abort()_i$ events; (2) every two $bcast()_i$ events have an intervening $ack()_i$ or $abort()_i$ event.

Cause function. There exists a "cause" function that maps each $rcv(m)_j$ event to a preceding $bcast(m)_i$ event, for $i \neq j$, and maps each $ack(m)_i$ event and each $abort(m)_i$ event to a preceding $bcast(m)_i$ event.

Constraints on message behavior. If an event $e \equiv bcast(m)_i$ "causes" an event $e' \equiv rcv(m)_j$, then (1) no other $rcv(m)_j$ event "caused" by e precedes e', and (2) no $ack(m)_i$ event caused by e precedes e'. If an event $e \equiv bcast(m)_i$ "causes" an event $e' \equiv ack(m)_i$, then (1) no other $ack(m)_i$ event "caused" by e precedes e', and (2) no $abort(m)_i$ event "caused" by e precedes e'. Finally, every $bcast(m)_i$ event "causes" either an $ack(m)_i$ event or an $abort(m)_i$ event.

Delay Functions. For time-complexity analysis, we assume that for every pair of processes i and j that are within the communication radius of each other, there exist upper bounds on the duration between a $bcast(m)_i$ event and the $rcv(m)_j$ and $ack(m)_i$ events "caused" by the $bcast(m)_i$ event. These upper bounds may depend on the communication graph G and are specified by two functions f_{rcv} and f_{ack}. The function f_{rcv} denotes the finite upper bound on the duration between a $bcast(m)_i$ event and the $rcv(m)_j$ event that is "caused" by event $bcast(m)_i$. The function f_{ack} denotes the upper bound on the duration between a $bcast(m)_i$ and the $ack(m)_i$ event that it "causes".

However, note that the algorithms presented in this paper do not use the bounds f_{rcv} and f_{ack}. The bounds are used only for time complexity analysis, and therefore, these bounds need not be known to the processes.

4. LEADER ELECTION

In this section we define the leader election (LE) problem and provide a solution to the LE problem in single-hop networks using the abstract MAC layer.

4.1 Problem Definition

Briefly, leader election (LE) is a problem in which all the processes in the system elect some process as the leader unanimously. A solution to LE is an I/O automaton [18] that has output actions $leader(l)_i$ for each process i and each process ID l, and the automaton has no input actions. In every fair execution, each process i elects a leader exactly once via an event $leader(l)_i$; furthermore, all processes elect the same leader l. More precisely, the LE problem is specified by two sets of properties of automaton executions: safety and liveness properties.

Safety Properties. We consider two safety properties, which hold in all executions. First, for each process i at most one $leader()_i$ event occurs. Second, for every pair of processes (i, j), if events $leader(l_i)_i$ and $leader(l_j)_j$ occur, then $l_i = l_j$ and l_i is the ID of some process in the system.

Liveness Property. The liveness property states that in any fair execution[1], for every process i in the system, some event of the form $leader()_i$ occurs.

[1]Note that a *fair execution* is one in which every process takes infinitely many steps. If no actions are enabled, then we assume that a process takes a no-op step.

4.2 Algorithm Overview

The pseudocode of our proposed LE algorithm is shown in Algorithm 1 and described next.

Algorithm 1: Leader election in 1-hop networks on top of the Abstract MAC layer.

```
Actions at each process $i$ whose ID is stored in the variable $id$.

Variables:
  $id$: a constant, contains the value $id_i$
  $output$: a process ID or $\perp$, initially $\perp$
  $neighbor$: a set of process IDs, initially $\emptyset$
  $candidate$: a set of process IDs, initially $\emptyset$
  $dropout$: a set of process IDs, initially $\emptyset$
  $IDset$: a set of process IDs, initially $\emptyset$

Actions:
  thread main
/* Phase 1: Discovery Phase */
  perform $bcast(\langle id, \text{"hello"}\rangle)$
  wait until event $ack(\langle id, \text{"hello"}\rangle)$ /* See thread ackHello */

/* Phase 2: Competition phase */
  if ($neighbor = \emptyset \vee id > \max(neighbor)$)
    then $msg := \langle id, \text{"compete"}\rangle$
  else $msg := \langle id, \text{"dropout"}\rangle$
  perform $bcast(msg)$
  while (($neighbor \not\subseteq dropout \cup candidate$)$\vee$
        (event $ack(msg)$ has not occurred))
    upon event $rcv(\langle id_j, \text{"dropout"}\rangle)$:
      $dropout := dropout \cup \{id_j\}$
    upon event $rcv(\langle id_j, \text{"compete"}\rangle)$:
      $candidate := candidate \cup \{id_j\}$
  endwhile
/* Event ack(msg) has occured before exiting the loop */

/* Phase 3: Decision phase */
  if (($neighbor = \emptyset$) $\vee$ ($id > \max(neighbor)$)$\wedge$
     ($dropout \cap neighbor = neighbor$))
    perform $bcast(\langle id, \text{"leader"}\rangle)$
    wait until event $ack(\langle id, \text{"leader"}\rangle)$
    $output := id$
  else
    wait until event $rcv(\langle id_j, \text{"leader"}\rangle)$
    $output := id_j$
  perform $leader(output)$

 thread recvHello
   upon event $rcv(\langle id_j, \text{"hello"}\rangle)$: $IDset := IDset \cup \{id_j\}$

 thread ackHello
   upon event $ack(\langle id, \text{"hello"}\rangle)$: $neighbor := IDset$
```

The algorithm consists of three phases at each process: *discovery* phase (Phase 1), *competition* phase (Phase 2), and *decision* phase (Phase 3).

In the discovery phase, each process i broadcasts a "hello" message via the abstract MAC layer's $bcast()_i$ action and waits for the ack of its broadcast before moving to the competition phase. Note that the abstract MAC layer guarantees that i will receive a unique ack for the broadcast, and furthermore, all the other processes within i's communication radius will receive the "hello" message before i receives the ack. The set of processes from whom i receives a "hello" message, prior to the ack of its own broadcast, constitutes the *neighbor set* of process i. Note that it is possible for some process to receive the ack of its "hello" message before having received "hello" messages from all or any of the other processes. Therefore, the neighbor set need not include all of the processes in the network.

In the competition phase, if every process in i's neighbor set has a lower ID than i, then i broadcasts a "compete" message; otherwise, i drops out of the competition and broadcasts a "dropout" message. In both cases, i broadcasts its message via the abstract MAC layer's $bcast()_i$ action. Upon receiving a "compete" (or "dropout") message from j, i adds j to its *candidate set* (or, *dropout set*, respectively). Process i remains in the competition phase until two conditions are satisfied: (1) process i has received a "compete" or a "dropout" broadcast message from each of the processes in its *neighbor* set, and (2) process i has received the ack (from the abstract MAC layer) for its sole broadcast in the competition phase. When the above two conditions are satisfied, i transitions to the decision phase.

In the decision phase, process i elects itself to be the leader iff i was competing in the competition phase and all the processes in its *neighbor* set are in its *dropout* set. Upon electing itself leader, process i broadcasts a "leader" message via the $bcast()_i$ action. Otherwise, i waits for some other process to broadcast a "leader" message and elects the sender of that message as the leader.

4.3 Proof of Correctness

To show that the pseudocode in Alg. 1 solves LE, we need to show that exactly one process broadcasts the "leader" message in the decision phase. First, we show that at the end of the discovery phase, for every pair of processes i and j, either $id_i \in neighbor_j$ or $id_j \in neighbor_i$. Moreover, for some set C of processes, their *neighbor* set consists of IDs strictly less than their own ID. Next, we note that all the processes in C compete in the competition phase. Finally, we show that the process with the smallest ID in C is the unique process that elects itself the leader in the decision phase.

Fix a fair execution α of the system composed of Alg. 1, the abstract MAC layer automaton, and the physical network.

LEMMA 1. *In α, for every process i at most one $leader()_i$ event occurs.*

PROOF. Follows from the pseudocode. $\square$

LEMMA 2. *Let i and j be a pair of processes. At least one of the following is true in the suffix of α that follows events $ack(\langle id_i, \text{"hello"}\rangle)_i$ and $ack(\langle id_j, \text{"hello"}\rangle)_j$: (1) $id_i \in neighbor_j$, (2) $id_j \in neighbor_i$.*

PROOF. From the pseudocode we know that the event $bcast(\langle id_i \text{"hello"}\rangle)_i$ and the event $bcast(\langle id_j \text{"hello"}\rangle)_j$ occur exactly once at indices (say) t_i and t_j of α, respectively. From the pseudocode, we know that α satisfies the well-formedness properties described in Section 3.1. Therefore, from the properties of the MAC layer, we know that, in α, the event $ack(\langle id_i, \text{"hello"}\rangle)_i$ occurs at index (say) $t_{i.a}$ (subscript a stands for "ack") and the event $ack(\langle id_j, \text{"hello"}\rangle)_j$ occurs at index $t_{j.a}$, respectively. Also, from the MAC layer properties, we know that, in α, event $rcv(\langle id_j, \text{"hello"}\rangle)_i$ occurs at index (say) $t_{i.r}$ (subscript r stands for "rcv") and event $rcv(\langle id_i, \text{"hello"}\rangle)_j$ occurs at index (say) $t_{j.r}$, respectively. Furthermore, $t_i < t_{j.r} < t_{i.a}$ and $t_j < t_{i.r} < t_{j.a}$, because of the guarantees of the MAC layer.

If $t_{j.a} < t_{i.a}$, then we know that $t_{i.r} < t_{i.a}$, else we know that $t_{j.r} < t_{j.a}$. Thus, without loss of generality, let $t_{i.r} < t_{i.a}$ in α. That is, when $rcv(\langle id_j, \text{"hello"}\rangle)_i$ occurs (at time $t_{i.r}$), the event $ack(\langle id_i, \text{"hello"}\rangle)_i$ has not occurred yet; therefore, when event $ack(\langle id_i, \text{"hello"}\rangle)_i$ occurs, i adds id_j to $neighbor_i$. Finally, from the pseudocode, we see that id_j is never removed from $neighbor_i$. $\square$

In α, let C (standing for "competitors") denote the set of process IDs such that for each $id_i \in C$, when process i starts executing Phase 2, either $neighbor_i = \emptyset$ or $id_i > \max(neighbor_i)$. Next, we show that $C \neq \emptyset$; that is, there exists at least one such process.

LEMMA 3. *In α, $C \neq \emptyset$.*

PROOF. Since the IDs at all processes are unique and are picked from a totally ordered namespace, we know that there exists a process whose ID is the largest among all the other processes in the system. Let that process be i (its ID is id_i). From the pseudocode, we know that the set $neighbor_i$ is populated by the IDs of other processes in the system. Since i has the largest ID in the system, it follows that either $neighbor_i = \emptyset$ or $id_i > \max(neighbor_i)$. Therefore, $id_i \in C$ and $C \neq \emptyset$. $\square$

From Lemma 3, we know that $\min(C)$, which denotes the smallest ID in C, exists. From the pseudocode we know that for each process i, the set $neighbor_i$ remains unchanged after the event $ack(\langle id_i, \text{"hello"}\rangle)_i$. Hereafter, when we refer to the value of $neighbor_i$, we refer to this unchanging value for each process i.

LEMMA 4. *In α, if $id_i = \min(C)$ then $neighbor_i \cap C = \emptyset$.*

PROOF. Let j be an arbitrary process such that $id_j \in neighbor_i$. Since $id_i \in C$, it follows that $id_i > id_j$. Also, $id_i = \min(C)$, so by the definition of $\min(C)$, it follows that $id_j \notin C$. In other words, $neighbor_i \cap C = \emptyset$. $\square$

LEMMA 5. *In α, for each $id_c \in C$, events $bcast(\langle id_c, \text{"compete"}\rangle)_c$ and $ack(\langle id_c, \text{"compete"}\rangle)_c$ occur; furthermore, for each $id_d \notin C$, events $bcast(\langle id_d, \text{"dropout"}\rangle)_d$ and $ack(\langle id_d, \text{"dropout"}\rangle)_d$ occur.*

PROOF. From the definition of C, we know that for each process c, where $id_c \in C$, when c starts Phase 2, $neighbor_c = \emptyset$ or $id_c > \max(neighbor_c)$. Therefore, in Phase 2 of the pseudocode, we see that for each process c, the event $bcast(\langle id_c, \text{"compete"}\rangle)_c$ occurs exactly once. From the properties of the abstract MAC layer, we know that the event $ack(\langle id_c, \text{"compete"}\rangle)_c$ occurs exactly once. From the pseudocode, we know that events $bcast(\langle id_c, \text{"dropout"}\rangle)_c$ and $ack(\langle id_c, \text{"dropout"}\rangle)_c$ do not occur.

Similarly, from the definition of C, we know that for each process ID $id_d \notin C$, when d starts Phase 2, $neighbor_d \neq \emptyset$ and $id_d < \max(neighbor_d)$. Therefore, in Phase 2, for each process d, event $bcast(\langle id_d, \text{"dropout"}\rangle)_d$ occurs exactly once. From the properties of the abstract MAC layer, event $ack(\langle id_d, \text{"dropout"}\rangle)_d$ occurs exactly once. From the pseudocode, we know that events $bcast(\langle id_d, \text{"compete"}\rangle)_d$ and $ack(\langle id_d, \text{"compete"}\rangle)_d$ do not occur. $\square$

LEMMA 6. *In α, if $id_i = \min(C)$, then at each process j, the event $leader(id_i)_j$ occurs.*

PROOF. Let $id_i = min(C)$ and j be an arbitrary process. We consider three cases: (1) $j = i$, (2) $j \neq i$ and $id_j \in neighbor_i$, and (3) $j \neq i$ and $id_j \notin neighbor_i$.

Case 1: $j = i$. Since $id_j = id_i = \min(C)$, from Lemma 5, it follows that event $bcast(\langle id_j, \text{"compete"}\rangle)_j$ and event $ack(\langle id_j, \text{"compete"}\rangle)_j$ occur. From Lemma 4, we know that for each $id_k \in neighbor_j$, $id_k \notin C$. Therefore, by Lemma 5, event $bcast(\langle id_k, \text{"dropout"}\rangle)_k$ occurs, and from the properties of the MAC layer, event $rcv(\langle id_k, \text{"dropout"}\rangle)_j$ occurs. As a result, id_k is added to $dropout_j$ and not added to $candidate_j$. Therefore, $candidate_j \cap neighbor_j$ is always empty, and eventually, $dropout_j \cap neighbor_j = neighbor_j$. From the pseudocode, we know that no ID is deleted from the sets $candidate_j$ and $dropout_j$. Therefore, eventually, event $bcast(\langle id_j, \text{"leader"}\rangle)_j$ occurs in the decision phase, and after event $ack(\langle id_j, \text{"leader"}\rangle)_j$ occurs, $output_j = id_j$.

Case 2: $j \neq i$ *and* $id_j \in neighbor_i$. From Lemma 4, we know that $id_j \notin C$. Therefore, applying Lemma 5, we know that event $bcast(\langle id_j, \text{"dropout"}\rangle)_j$ occurs. From the pseudocode, we see that event $bcast(\langle id_j, \text{"dropout"}\rangle)_j$ occurs when $neighbor_j \neq \emptyset$ and $id_j < \max(neighbor_j)$ in Phase 2. However, in Phase 3, since $neighbor_j \neq \emptyset$ and $id_j < \max(neighbor_j)$, it follows that event $bcast(\langle id_j, \text{"leader"}\rangle)_j$ does not occur.

Case 3: $j \neq i$ *and* $id_j \notin neighbor_i$. We know from Lemma 2 that $id_i \in neighbor_j$. From Lemma 5, we know event $bcast(\langle id_i, \text{"compete"}\rangle)_i$ occurs, and from the properties of the abstract MAC layer, event $rcv(\langle id_i, \text{"compete"}\rangle)_j$ occurs. Consequently, $dropout_j \cap neighbor_j \neq neighbor_j$. Therefore, event $bcast(\langle id_j, \text{"leader"}\rangle)_j$ does not occur.

Thus, from Cases 1, 2 and 3, it follows that the event $bcast(\langle id_j, \text{"leader"}\rangle)_j$ occurs only when $j = i$ (that is, only for process i where $id_i = \min(C)$). From the properties of the MAC layer, we know that for each process $j \neq i$, event $rcv(\langle id_i, \text{"leader"}\rangle)_j$ occurs. Therefore, for each process j in the system, in the decision phase $output_j = id_i$. Since no other event in α changes the value of $output$, from the pseudocode we know that the event $leader(id_i)_j$ occurs. □

THEOREM 7. *The pseudocode in Algorithm 1 solves leader election when composed with the abstract MAC layer and the physical network.*

PROOF. Consider an arbitrary execution α of the system composed of Algorithm 1, the abstract MAC layer automaton, and the physical network. From Lemma 1, we know that at each process j, at most one $leader()_j$ event occurs in α. From Lemma 6 we know that, in α, for each process j in the system, event $leader(id_i)_j$ occurs where i is some unique process in the system. Thus, the safety and liveness properties of leader election are satisfied. □

4.4 Message and Time Complexity

In any execution of Algorithm 1 with an abstract MAC layer, we see from the pseudocode that each process i sends exactly three messages using the abstract MAC layer interface. For time complexity, we assume synchronous wake up, zero local-step time, and consider the delay functions of the abstract MAC layer to compute the time complexity. Let F_{ack} be the maximum of all f_{ack} values of all processes. Algorithm 1 composed with an abstract MAC layer terminates within $3F_{ack}$ time units because each process sends and waits for the ack for exactly three messages.

5. MAXIMAL INDEPENDENT SET

In this section we define the problem of computing a maximal independent set (MIS) and provide an algorithm which computes an MIS in a multi-hop network using the abstract MAC layer.

5.1 Problem Definition

A maximal independent set (MIS) of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that for every vertex $v_i \in V$ either $v_i \in S$ or there exists an edge $(v_i, v_j) \in E$ such that $v_j \in S$, and for every edge $(v_i, v_j) \in E$, $\{v_i, v_j\} \not\subseteq S$. A solution to MIS is an I/O automaton [18] whose output actions are $member(b)_i$ for each process i and $b \in \{true, false\}$, and it has no input actions. In any fair execution, the automaton eventually outputs events $member(b_i)_i$ (exactly once) at each process i where b_i is a Boolean such that the set S of processes at which the event $member(true)$ occurs is an MIS. More precisely, the MIS problem is specified by two sets of properties of automaton executions: safety and liveness properties.

For each process i, let N_i denote the set consisting of all the processes that are neighbors of i in G.

Safety Properties. First, in any execution, for each process i, at most one $member()_i$ event occurs. Second, the following two properties must hold:

- *Independence:* In any execution, for every pair of processes i and j that are neighbors, if events $member(b_i)_i$ and $member(b_j)_j$ occur, then it is not the case that $b_i = b_j = true$.
- *Maximality:* In any execution, for every process i, and for all $j' \in N_i \cup \{i\}$, if events $member()_{j'}$ occur, then $\exists j \in N_i \cup \{i\}$ such that event $member(true)_j$ occurs.

Liveness Property. In any fair execution, for each process i, some $member()_i$ event occurs.

5.2 Algorithm Overview

The pseudocode of the algorithm is shown in Algorithm 2. The algorithm consists of three phases at each process: *discovery* phase, *competition* phase, and *decision* phase. Each process announces itself to all its neighbors (the processes that are within its communication radius) in the discovery phase. In the competition phase, each process participates in a competition with its neighbors to determine whether or not it should join the MIS. In the decision phase, processes that win the competition join the MIS, and other process terminate without joining the MIS. These three phases are part of thread main of the pseudocode of the algorithm.

Each process, in its discovery phase, broadcasts its ID through a "hello" message. The phase ends when the process receives the ack for its broadcast, and the process proceeds to the competition phase. Throughout the execution, when a process (say) i receives a "hello" message from another process (say) j, the ID id_j is added to the set $neighbor_i$ (as shown in thread rcvHello in the pseudocode). Unlike in Algorithm 1, here i updates its $neighbor_i$ set throughout the execution of the algorithm.

In the competition phase, each process competes or drops out depending on the IDs in its $neighbor_i$ set. Each process i, at the start of its competition phase, is neither competing nor dropping out. The competition phase consists of a *while* loop and each process i remains in this loop until it can irrevocably decide on its membership in the MIS.

Algorithm 2: MIS in multi-hop networks with the Abstract MAC layer.

```
Actions at each process $i$ whose ID is $id$.

Variables:
  $currentState$: an element of $\{hello, compete, droppedOut\}$,
                  initially $hello$
  $quit$: a Boolean, initially $false$
  $neighbor$: a set of process IDs, initially $\emptyset$
  $candidate$: a set of process IDs, initially $\emptyset$
  $dropout$: a set of process IDs, initially $\emptyset$

Actions:
  thread rcvHello
    upon event $rcv(\langle id_j, \text{"hello"}\rangle)$:
       $neighbor := neighbor \cup \{id_j\}$

  thread rcvCompete
    upon event $rcv(\langle id_j, \text{"compete"}\rangle)$:
      $candidate := candidate \cup \{id_j\}$
      $dropout := dropout \setminus \{id_j\}$

  thread rcvDropout
    upon event $rcv(\langle id_j, \text{"dropout"}\rangle)$:
      $dropout := dropout \cup \{id_j\}$
      $candidate := candidate \setminus \{id_j\}$

  thread rcvMember
    upon event $rcv(\langle id_j, \text{"member"}\rangle)$:
      $quit := true$

  thread rcvNotMember
    upon event $rcv(\langle id_j, \text{"not member"}\rangle)$:
      $neighbor := neighbor \setminus \{id_j\}$
      $candidate := candidate \setminus \{id_j\}$
      $dropout := dropout \setminus \{id_j\}$

  thread main /* See Algorithm 3 */

cobegin /* Execute the following threads concurrently */
  main // rcvHello // rcvCompete // rcvDropout //
  rcvNotMember // rcvMember
coend
```

Each iteration of the *while* loop may be viewed as a local phase of the algorithm. Different processes may execute a different number of such local phases, and different processes may be at different phases concurrently. At the start of the loop, each process i determines whether it will compete or drop out of competition based on the following rule: if some process ID in the $neighbor_i$ set has a higher value that id_i, then process i drops out of the competition (because a higher-ID neighbor may join the MIS), and otherwise, process i competes. In either case, every time process i changes its decision to compete, or to drop out, it broadcasts a message announcing its intention (to either compete or drop out). Also, process i maintains a partitioning of the $neighbor_i$ set into two disjoint sets: $candidate_i$ and $dropout_i$. The set $candidate_i$ contains the processes that are competing to join the MIS, and the set $dropout_i$ contains the processes that are dropped out of competition.

The $candidate_i$ and $dropout_i$ sets are updated as follows. Upon receiving a message from some other process (say) j announcing that j is competing, id_j is added to $candidate_i$ and removed from $dropout_i$ (as shown in thread rcvCompete of the pseudocode). Similarly, upon receiving a message from j announcing that it is dropped out, id_j is added to the $dropout_i$ set and removed from the $candidate_i$ set (as shown in thread rcvDropout of the pseudocode). Note that a process i may compete in one iteration (because it has the highest ID among the neighbors it has heard from) and drop out in another iteration (because it received a message from a neighbor with a higher ID than id_i) and revert to competing again in some future iteration (because all the neighbors with higher ID than id_i have decided not to join the MIS). The number of times process i switches back and forth between competing and dropping out depends on the number of neighbors of i with ID higher than id_i.

Algorithm 3: Thread *main* from Algorithm 2.

```
thread main
 /* Phase 1: Discovery Phase */
   perform $bcast(\langle id_i, \text{"hello"}\rangle)$
   wait until event $ack(\langle id_i, \text{"hello"}\rangle)$

 /* Phase 2: Competition Phase */
   while $((neighbor \neq dropout \vee currentState \neq compete) \wedge$
           $\neg quit)$
     if $(neighbor = \emptyset \vee id_i > \max(neighbor))$
       if $(currentState \neq compete)$
         perform $bcast(\langle id_i, \text{"compete"}\rangle)$
         wait until $ack(\langle id_i, \text{"compete"}\rangle)$
         $currentState := compete$
     else
       if $(currentState \neq droppedOut)$
         perform $bcast(\langle id_i, \text{"dropout"}\rangle)$
         wait until $ack(\langle id_i, \text{"dropout"}\rangle)$
         $currentState := droppedOut$
     wait until $neighbor = candidate \cup dropout$
   endwhile

 /* Phase 3: Decision Phase */
   if $(quit)$ /* Do not join MIS */
     perform $bcast(\langle id_i, \text{"not member"}\rangle)$
     wait until event $ack(\langle id_i, \text{"not member"}\rangle)$
     perform $member(false)$
   else /* Join MIS */
     perform $bcast(\langle id_i, \text{"member"}\rangle)$
     wait until event $ack(\langle id_i, \text{"member"}\rangle)$
     perform $member(true)$
   halt
```

A process can irrevocably join the MIS iff all of its neighbors have either dropped out or decided not to join the MIS, and a process can irrevocably not join the MIS iff one of its neighbors joins the MIS. Eventually, each process i exits the while loop, and enters the decision phase, if (1) $neighbor_i$ is empty, (2) process i is competing whereas all the processes in its $neighbor_i$ set have dropped out, or (3) one of its neighbors joins the MIS. In cases (1) and (2), i joins the MIS, and in case (3), i decides to not join the MIS.

If a process i receives a broadcast from j that states that j has joined the MIS, then process i decides not to join the MIS (through setting the *quit* variable to true in thread rcvMember). If a process i receives a broadcast from j that states that j has decided to not join the MIS (through thread rcvNotMember), then i removes id_j from its $neighbor_i$ set as shown in thread rcvNotMember (and therefore, id_j is removed from the $dropout_i$ and $candidate_i$ sets as well).

5.3 Proof of Correctness

In order to show that the pseudocode in Algorithm 2 computes an MIS, we need to show that all fair executions of Algorithm 2 satisfy the safety and liveness properties of MIS in Section 5.

The structure of the proof of correctness is as follows. First, note that the pseudocode establishes the first safety requirement: for each process i, at most one $member()_i$ event occurs. The lemmas that follow help establish the independence and maximality safety properties, and finally, the progress property.

Lemma 8 and Corollary 9 establish that while a process i is in its competition phase, for all neighbors j of i, id_i is in j's *neighbor* set. If process i (eventually) joins the MIS, then the following Lemmas hold. Lemma 10 shows that for all neighbors j of i, id_i is in the $neighbor_j$ set even after i exits the competition phase. Lemma 11 establishes that when i exits its competition phase, its $quit_i$ variable is $false$, and Lemma 12 shows that for the remainder of the execution, for each neighbor j of i, id_i is in the $candidate_j$ set.

The above lemmas help establish the independence property in Lemma 13, and the maximality property in Lemma 14. The liveness property is established through helper Lemmas 15 and 16: Lemma 15 shows that at each process, its *neighbor* set is always a superset of the union of its *dropout* and *candidate* set, and Lemma 16 shows that at each process, its *neighbor* set is eventually and permanently equal to the union of its *dropout* and *candidate* set. Lemma 16 is used to establish the liveness property in Lemma 17 and Corollary 18. Finally, Theorem 19 proves correctness by invoking Lemmas 13 and 14, and Corollary 18.

For the remainder of this section, fix a fair execution α of the automaton composed of Algorithm 2, the abstract MAC layer and the physical system. Let $\alpha[k]$ denote the k-th event in α; k is said to be the *index* at which event $\alpha[k]$ occurs.

LEMMA 8. *In α, for every pair of processes (i, j), where i and j are neighbors, in the suffix that follows the event $ack(\langle id_i, \text{"hello"}\rangle)_i$, $id_i \in neighbor_j$, until the event $rcv(\langle id_i, \text{"not member"}\rangle)_j$ occurs.*

PROOF. From the pseudocode we know that the event $bcast(\langle id_i, \text{"hello"}\rangle)_i$ occurs exactly once in α at index (say) t_i. That is, $\alpha[t_i]$ denotes the event $bcast(\langle id_i, \text{"hello"}\rangle)_i$. From the properties of the MAC layer, we know that, in α, the event $ack(\langle id_i, \text{"hello"}\rangle)_i$ occurs at index (say) $t_{i.a}$ (subscript a stands for "ack"). Also, from the properties of the MAC layer, we know that, in α, event $rcv(\langle id_i, \text{"hello"}\rangle)_j$ occurs at index (say) $t_{j.r}$ (subscript r stands for "rcv"), where $t_i < t_{j.r} < t_{i.a}$. From the pseudocode, we know that j adds id_i to $neighbor_j$ in thread rcvHello when event $rcv(\langle id_i, \text{"hello"}\rangle)_j$ occurs. Finally, note that id_i is removed from $neighbor_j$ only in thread rcvNotMember when event $rcv(\langle id_i, \text{"not member"}\rangle)_j$ occurs. $\square$

COROLLARY 9. *In α, while process i is in its competition phase, for each process j that is a neighbor of i, $id_i \in neighbor_j$.*

PROOF. Follows from the pseudocode, Lemma 8, and the observation that event $rcv(\langle id_i, \text{"not member"}\rangle)_j$ occurs only when i is in its decision phase. $\square$

LEMMA 10. *In α, if event $member(true)_i$ occurs, then in the suffix of α starting from the last state in which process i is in its competition phase, for each neighbor j of i, $id_i \in neighbor_j$.*

PROOF. From Lemma 8, we know that in the suffix following event $ack(\langle id_i, \text{"hello"}\rangle)_i$ (which occurs in the discovery phase), for each process j that is a neighbor of i, $id_i \in neighbor_j$ until event $rcv(\langle id_i, \text{"not member"}\rangle)_j$ occurs. However, from the pseudocode we know that such an event cannot occur if $member(true)_i$ event occurs. Therefore, the suffix of α starting from when i enters its competition phase $i \in neighbor_j$. $\square$

LEMMA 11. *In α, if event $member(true)_i$ occurs, then in all the states in α where process i is in its competition phase, $quit_i = false$.*

PROOF. If event $member(true)_i$ occurs, then from the pseudocode, we see that in the state succeeding the event in which i enters its decision phase, $quit_i = false$. From the pseudocode, we know that before entering its decision phase, i is in its competition phase. Since we know from the pseudocode that $quit_i = true$ is a stable predicate, for $quit_i$ to be $false$ when i enters its decision phase, $quit_i$ must be $false$ while i is in its competition phase. $\square$

LEMMA 12. *In α, if event $member(true)_i$ occurs, then in the suffix of α starting from the last state in which process i is in its competition phase, for each neighbor j of i, $id_i \in candidate_j$.*

PROOF. Applying Lemma 11 we know that, when i exits the *while* loop in the main thread, $quit_i$ is false. However, the exit condition for the loop is that either (1) $quit_i$ is $true$, or (2) $neighbor_i = dropout_i$ and $currentState_i = compete$. Thus, we know that when i exits the *while* loop, $neighbor_i = dropout_i$ and $currentState_i = compete$.

From the pseudocode, we see that in order for the value of $currentState_i$ to be set to $compete$ upon exiting the *while* loop, the following must be true. Let events $\alpha[t_{bcast}] \equiv bcast(\langle id_i, \text{"compete"}\rangle)_i$ and $\alpha[t_{ack}] \equiv ack(\langle id_i, \text{"compete"}\rangle)_i$ be the last pair of $bcast()_i$ and $ack()_i$ events to occur before i exits the *while* loop. Therefore, no $bcast(\langle id_i, \text{"dropout"}\rangle)_i$ event occurs after event $\alpha[t_{ack}]$. The above argument concludes that for each process j that is a neighbor of i, event $\alpha[t_{j.rcv}] \equiv rcv(\langle id_i, \text{"compete"}\rangle)_j$ occurs which is "caused" by $\alpha[t_{bcast}]$ and no event $rcv(\langle id_i, \text{"dropout"}\rangle)_j$ occurs after $\alpha[t_{j.rcv}]$. Therefore, we conclude that event $\alpha[t_{j.rcv}]$ adds id_i to the set $candidate_j$ and id_i is never removed from $candidate_j$ thereafter.

By the properties of the abstract MAC layer, we know that $t_{j.rcv} < t_{ack}$. Also, event $\alpha[t_{ack}]$ occurs before i exits the competition phase. Therefore, if event $member(true)_i$ occurs, then in the suffix of α starting from the last state in which process i is in its competition phase, for each neighbor j of i, $id_i \in candidate_j$. $\square$

LEMMA 13. *In α, for each pair of processes (i, j), where i and j that are neighbors, if event $member(b_i)_i$ and event $member(b_j)_j$ occur, then it is not the case that $b_i = b_j = true$.*

PROOF. For the purpose of contradiction, assume that $b_i = b_j = true$. Applying Lemmas 10 and 12 to i and its neighbor j, we get the following: in the suffix of α starting from the last state S_i in which process i is in its competition phase, $id_i \in neighbor_j \cap candidate_j$. Let $\alpha[t_i]$ denote the event in α immediately succeeding S_i.

Applying Lemmas 10 and 12 to j and its neighbor i, we see that in the suffix of α starting from the last state S_j in which process j is in its competition phase, $id_j \in neighbor_i \cap candidate_i$. Let $\alpha[t_j]$ denote the event in α immediately succeeding S_j.

Since S_j is the last state in α in which j is in the competition phase, the exit condition for the *while* loop must be enabled in S_j. Therefore, either (1) $quit_j = true$ or (2) $neighbor_j = dropout_j$. Since j is in the competition phase in S_j, applying Lemma 11 we know that $quit_j = false$. Therefore, $neighbor_j = dropout_j$ in S_j.

Without loss of generality, let $t_i < t_j$. Therefore, S_j is in the suffix of α that follows event $\alpha[t_i]$, and consequently, in S_j, $id_i \in neighbor_j \cap candidate_j$. From the pseudocode, we see that $id_i \notin dropout_j$ in S_j. Thus, we see that in S_j, $id_i \in neighbor_j$, and $id_i \notin dropout_j$; this contradicts our earlier conclusion that $neighbor_j = dropout_j$ in S_j. □

LEMMA 14. *In α, for every process i, and for all $j' \in N_i \cup \{i\}$, if events $member()_{j'}$ occur, then $\exists j \in N_i \cup \{i\}$ such that event $member(true)_j$ occurs.*

PROOF. Suppose in contradiction that there is no $j \in N_i \cup \{i\}$ such that $member(true)_j$ occurs. Therefore, for all processes $j \in N_i \cup \{i\}$, event $member(false)_j$ occurs. In particular, event $member(false)_i$ occurs.

From the pseudocode we know that event $member(false)_i$ occurs if $quit_i = true$. Note that $quit_i$ is set to $true$ (in thread rcvMember) only when, for some process $j \in N_i$, event $rcv(\langle id_j, \text{“member”}\rangle)_i$ occurs. This implies that event $bcast(\langle id_j, \text{“member”}\rangle)_j$ must have occurred in α. From the pseudocode, we know that event $bcast(\langle id_j, \text{“member”}\rangle)_j$ is followed by event $member(true)_j$. Therefore, if the event $member(false)_i$ occurs in α, then for some process j that is a neighbor of i, the event $member(true)_j$ occurs: a contradiction. □

LEMMA 15. *In α, for each process i, the following invariant holds: $neighbor_i \supseteq dropout_i \cup candidate_i$.*

PROOF. Fix a process i. The proof is by induction on the number of events in α. For the base case, note that, initially, $neighbor_i = dropout_i = candidate_i = \emptyset$. Therefore, in the initial state of α, $neighbor_i \supseteq dropout_i \cup candidate_i$.

For the inductive hypothesis, assume that for some state S in α, $neighbor_i \supseteq dropout_i \cup candidate_i$. In the inductive step, we show that in the state following S in α, $neighbor_i \supseteq dropout_i \cup candidate_i$.

Let the state following S in α be S'. If the event e between S and S' occurs at a process $j \neq i$, then we see that $neighbor_i \supseteq dropout_i \cup candidate_i$ in S', and the inductive step is complete. For the remainder of this proof, we assume that the event e occurs at process i.

The only events at process i that change the values of $neighbor_i$, $dropout_i$, or $candidate_i$ are in threads rcvHello, rcvNotMember, rcvCompete, and rcvDropout. We consider each thread separately.

Case 1. In thread rcvHello elements are only added to the set $neighbor_i$. Therefore, if $neighbor_i \supseteq dropout_i \cup candidate_i$ in S, then in S' resulting from executing thread rcvHello $neighbor_i \supseteq dropout_i \cup candidate_i$.

Case 2. Thread rcvNotMember removes the same element from all three sets $neighbor_i$, $dropout_i$, and $candidate_i$. Therefore, if $neighbor_i \supseteq dropout_i \cup candidate_i$ in S, then in S', resulting from executing thread rcvNotMember, it follows that $neighbor_i \supseteq dropout_i \cup candidate_i$.

Case 3. Thread rcvCompete adds a process ID (say) id_j to $candidate_i$ and removes it from $dropout_i$. If we show that $id_j \in neighbor_i$ in state S, then we know that in S', $neighbor_i \supseteq dropout_i \cup candidate_i$, and the inductive step is complete.

Note that thread rcvCompete is executed only when event $e_{rcv} \equiv rcv(\langle id_j, \text{“compete”}\rangle)_i$ occurs. From the properties of the abstract MAC layer we know that there exists a preceding event $e_{bcast} \equiv bcast(\langle id_j, \text{“compete”}\rangle)_j$ that “caused” e_{rcv}, and there also exists a succeeding event $e_{ack} \equiv ack(\langle id_j, \text{“compete”}\rangle)_j$ that was “caused” by e_{bcast}. From the pseudocode we know that in the execution segment from e_{bcast} to e_{ack}, process j is in its competition phase. Therefore, applying Corollary 9, we know that $id_j \in neighbor_i$ in states S and S'.

Case 4. Thread rcvDropout adds a process ID (say) id_j to $dropout_i$ and removes it from $candidate_i$. The argument for inductive step in this case is similar to Case 3 and has been omitted.

Thus, in α, for each process i, the following invariant holds: $neighbor_i \supseteq dropout_i \cup candidate_i$. □

LEMMA 16. *In α, for each process i, eventually and permanently, $neighbor_i = dropout_i \cup candidate_i$.*

PROOF. Assume for contradiction that there exists some process i such that $neighbor_i \neq dropout_i \cup candidate_i$, infinitely often in α. From Lemma 15, we know that throughout α, $neighbor_i \supseteq dropout_i \cup candidate_i$. Therefore, we know that $neighbor_i \supset dropout_i \cup candidate_i$, infinitely often in α. That is, there exists a process j such that $id_j \in neighbor_i$ and $id_j \notin dropout_i \cup candidate_i$, infinitely often.

From the pseudocode, we see that if id_j is deleted from $neighbor_i$ in some event e, then in the suffix of α that follows e, $id_j \notin neighbor_i$. Therefore, if $id_j \in neighbor_i$ infinitely often, then in all the states of some suffix of α, $id_j \in neighbor_i$.

Also, from the pseudocode, we see that if $id_j \in dropout_i \cup candidate_i$ in any state S in α, then no event removes id_j from $dropout_i \cup candidate_i$ without also removing id_j from $neighbor_i$. Consequently, if $id_j \in neighbor_i$ and $id_j \notin dropout_i \cup candidate_i$, infinitely often in α, then in all the states of α, $id_j \notin dropout_i \cup candidate_i$.

However, from the pseudocode, we see that process j eventually enters its competition phase, and from Corollary 9, we know that $id_j \in neighbor_i$. Also, in process j's competition phase, either $bcast(\langle id_j, \text{“compete”}\rangle)_j$ event or $bcast(\langle id_j, \text{“dropout”}\rangle)_j$ event occurs. From the properties of the abstract MAC layer, we know that such an event “causes” either an event $rcv(\langle id_j, \text{“compete”}\rangle)_i$ or an event $rcv(\langle id_j, \text{“dropout”}\rangle)_i$, respectively. Consequently, there exists a state S' in α in which either $id_j \in candidate_i$ or $id_j \in dropout_i$. This contradicts our earlier conclusion that $id_j \notin dropout_i \cup candidate_i$ throughout α. □

For any prefix of α, let M denote the set of processes at which some $member()$ event has occurred in that prefix. Recall that J is the set of processes in the system. Therefore, $M \subseteq J$ and $|M| \leq n$. We now prove progress through the following helper lemma.

LEMMA 17. *For all $k \in \mathbb{N}$, where $0 \leq k \leq n$, there exists a prefix β of α such that $|M| = k$.*

PROOF. The proof is by induction on the size of M.

Base Case: There exists a prefix β of α in which $|M| = 0$. The proof is as follows: initially, no $member()$ events have occurred, so $M = \emptyset$ and the base case holds true.

Inductive Hypothesis: There exists a prefix β of α such that $|M| = k$ for some $k < n$.

Inductive Step: Now we show that there exists a prefix β' of α in which $|M| = k + 1$.

Let γ be the suffix of α starting from the end of β. For the purpose of contradiction, assume that for each $j \in J \setminus M$, no $\mathit{member}()_j$ events occur in γ; that is, no $\mathit{member}()_j$ event occurs in α. Therefore, j is in its competition phase eventually and permanently. At the start of γ, let i be the process with the highest ID in $J \setminus M$ (so, no $\mathit{member}()_i$ events have occurred in β). Let N'_i denote the set of processes in $J \setminus M$ that are neighbors of i.

Since no $\mathit{member}()_i$ event occurs in α, i is in its competition phase eventually and permanently, and therefore, $\mathit{quit}_i = \mathit{false}$ in all states of α. From the pseudocode, we know that quit_i is a stable predicate. From the pseudocode, we see that quit_i is set to true when a $\mathit{rcv}(\langle id_j, \text{“member”}\rangle)_i$ event occurs (for some j). Therefore, for each process $j \in M$ that is a neighbor of i, event $\mathit{rcv}(\langle id_j, \text{“member”}\rangle)_i$ does not occur and event $\mathit{rcv}(\langle id_j, \text{“not member”}\rangle)_i$ occurs. Thus, we know that eventually and permanently, $id_j \notin \mathit{neighbor}_i$, $id_j \notin \mathit{dropout}_i$, and $id_j \notin \mathit{candidate}_i$. In other words, eventually and permanently, sets $\mathit{neighbor}_i$, $\mathit{dropout}_i$, and $\mathit{candidate}_i$ contain only the IDs of processes in N'_i.

Applying Corollary 9, we know that for each process $j \in N'_i$, $id_j \in \mathit{neighbor}_i$, and $id_i \in \mathit{neighbor}_j$. Since i has the highest ID in $N'_i \cup \{i\}$, eventually and permanently, either $\mathit{neighbor}_i = \emptyset$ or $id_i > \max(\mathit{neighbor}_i)$, and for each process $j \in N'_i$, $id_j < \max(\mathit{neighbor}_j)$ eventually and permanently. Therefore, from the pseudocode, we see that, for each $j \in N'_i$, eventually and permanently, $\mathit{currentState}_j = \mathit{droppedOut}$. Furthermore, we see that when $\mathit{currentState}_j$ changes to $\mathit{droppedOut}$ for the final time in α, the events $\mathit{bcast}(\langle id_j, \text{“dropout”}\rangle)_j$ and $\mathit{ack}(\langle id_j, \text{“dropout”}\rangle)_j$ occur, and until j exits the *while* loop, no additional $\mathit{bcast}()_j$ and $\mathit{ack}()_j$ events occur. Therefore, $\mathit{neighbor}_i = \mathit{dropout}_i$ eventually and permanently in α.

Recall that i has the highest ID in $N'_i \cup \{i\}$, and therefore, eventually and permanently, either $\mathit{neighbor}_i = \emptyset$ or $id_i > \max(\mathit{neighbor}_i)$. Therefore, eventually and permanently, $\mathit{currentState}_i = \mathit{compete}$. Note that $(\mathit{neighbor}_i = \mathit{dropout}_i) \wedge (\mathit{currentState}_i = \mathit{compete})$ is a sufficient condition for i to exit the *while* loop and enter its decision phase. In the decision phase, a $\mathit{member}()_i$ event occurs. This contradicts our assumption that for each process $j \in J \setminus M$, no $\mathit{member}()_j$ events occur in α. Therefore, some $\mathit{member}()$ event occurs in γ, and there exists a prefix β' of α in which $|M| = k + 1$.

Therefore, by induction we have proved that eventually $|M| = n$ in α; that is, for each process i, an event $\mathit{member}()_i$ occurs. $\square$

The following corollary establishes the liveness property.

Corollary 18. *For each process i, an $\mathit{member}()_i$ event occurs in α.*

Theorem 19. *The pseudocode in Algorithm 2 computes an MIS.*

Proof. Given any arbitrary admissible execution α of the automaton composed of Algorithm 2, the abstract MAC layer and the physical system, the pseudocode shows that for each process i at most one $\mathit{member}()_i$ event occurs, Lemmas 13 and 14 show that α satisfies the safety properties of MIS (independence and maximality, respectively), and Corollary 18 shows that α satisfies the liveness property of MIS. $\square$

5.4 Message and Time Complexity

In any execution of Algorithm 2 on an abstract MAC layer where Δ is the maximum degree of the communication graph, each process i sends at most $2\Delta+3$ messages through the abstract MAC layer interface. Intuitively, each neighbor j of i such that $id_j > id_i$ can contribute to an extra iteration of the *while* loop at i. Summing up the two messages per iteration, the initial “compete” message to enter the loop, the initial “hello” message and the final “member”/“not member” messages we get a total of $2\Delta + 3$ messages.

To compute the time complexity, we assume synchronous wakeup, zero local-step time, and consider the delay functions of the abstract MAC layer. Let F_{ack} be the maximum of all f_{ack} values of all processes. The pseudocode in Algorithm 2 computes an MIS within $(2n + 2)F_{ack}$ time. Intuitively, after processes are in the competition phase, in every $2F_{ack}$ time units, at least one process decides to join or not join the MIS. Summing up the initial F_{ack} time units for processes to enter the competition phase, $2F_{ack}$ time units for each process to make a decision, and additional F_{ack} time to broadcast the last “member”/“not member” message, we get a total of $(2n + 2)F_{ack}$ time units.

6. DISCUSSION

A Case for Asynchrony. Note that our leader election and MIS algorithms are asynchronous despite many wireless models and abstract MAC layer implementations providing timing guarantees. Such asynchronous algorithms offer two advantages. (1) The algorithms are agnostic of issues to synchrony and therefore deployable in any system on top of which the abstract MAC layer is constructed. (2) When deployed in systems with better synchrony guarantees, these algorithms automatically provide faster termination. For instance, our leader election algorithm uses a constant number of messages on the abstract MAC layer. When executed on the abstract MAC layer from [11], leader election can be solved in $\mathcal{O}(n \log n)$ time w.h.p. Better MAC layer implementations will automatically translate to better bounds for our algorithms.

High Time Complexity. The high time complexity $\mathcal{O}(n)$ of the MIS algorithm remains to be discussed. In contrast, the time complexities of other MIS algorithms are much lower. For instance, the randomized algorithm in [19] has a time complexity of $\mathcal{O}(\log^2 n)$, the deterministic algorithms in [25] have time complexities $\mathcal{O}(\log^2 n)$ without collision detection and $\mathcal{O}(\log n)$ with collision detection, the deterministic algorithm in [17] has a time complexity of $\mathcal{O}(\log \Delta \cdot \log^* n)$ (where Δ is the maximum degree of a process), and the deterministic algorithm in [24] has a time complexity of $\mathcal{O}(\log^* n)$. Next, we argue for the high time complexity of our algorithm despite the aforementioned results.

Although the system model in [19] is completely asynchronous without any collision detection, the MIS algorithm is randomized whereas our MIS algorithm is deterministic. An exponential gap between deterministic and randomized distributed algorithms is not unexpected [2, 6].

The algorithms in [17, 24, 25] are all deterministic, but assume lock-step synchrony. In such systems, deterministic neighborhood discovery requires just one round of broadcast, whereas in systems with asynchronous abstract MAC layer processes cannot deterministically 'know' their exact neighborhood.

Also, the algorithms in [17, 24, 25] are assumed to be executing in a growth bounded graph which imposes restrictions on the degree of the nodes and the node density in the wireless network. We make no such assumptions. We contend that the weak assumptions and the strong power of the adversary in our system model contributes to the high time complexity of our MIS algorithm.

7. REFERENCES

[1] H. Attiya and J. Welch. *Distributed Computing : Fundamentals, Simulations, and Advanced Topics.* John Wiley and Sons, Inc., 2004.

[2] R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in radio networks: an exponential gap between determinism randomization. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 98–108, 1987.

[3] J. L. Bordim, Y. Ito, and K. Nakano. Randomized leader election protocols in noisy radio networks with a single transceiver. In *Proc. of the 4th International Symposium on Parallel and Distributed Processing and Applications*, pages 246–256, 2006.

[4] J. I. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Trans. on Information Theory*, 25(5):505–515, 1979.

[5] A. Cornejo, N. Lynch, S. Viqar, and J. L. Jennifer L. Welch. Neighbor discovery in mobile ad hoc networks using an abstract MAC layer. In *Proc. of the 47th Annual Allerton Conference on Communication, Control, and Computing*, pages 1460–1467, 2009.

[6] M. Ghaffari, N. Lynch, and S. Sastry. Leader election using loneliness detection. In *Proc. of the 25th International Symposium on Distributed Computing*, pages 268–282, 2011.

[7] S. Gollakota and D. Katabi. Zigzag decoding: combating hidden terminals in wireless networks. In *Proc. of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 159–170, 2008.

[8] R. Ingram, P. Shields, J. Walter, and J. L. Welch. An asynchronous leader election algorithm for dynamic networks. In *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[9] T. Jurdziński, M. Kutylowski, and J. Zatopiański. Efficient algorithms for leader election in radio networks. In *Proc. of the 21st Annual Symposium on Principles of Distributed Computing*, pages 51–57, 2002.

[10] M. Khabbazian, F. Kuhn, D. R. Kowalski, and N. Lynch. Decomposing broadcast algorithms using abstract MAC layers. In *Proc. of the 6th International Workshop on Foundations of Mobile Computing*, pages 13–22, 2010.

[11] M. Khabbazian, F. Kuhn, N. Lynch, M. Medard, and A. P. Gheibi. MAC design for analog network coding. In *Proc. of the 7th ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, pages 42–51, 2011.

[12] D. Kowalski and A. Pelc. Broadcasting in undirected ad hoc radio networks. *Distributed Computing*, 18(1), 2005.

[13] D. Kowalski and A. Pelc. Leader election in ad hoc radio networks: A keen ear helps. In *Proc. of the 36th International Colloquium on Automata, Languages and Programming*, pages 521–533, 2009.

[14] F. Kuhn, N. Lynch, and C. Newport. The abstract MAC layer. In *Proc. of the 23rd International Conference on Distributed Computing*, pages 48–62, 2009.

[15] F. Kuhn, N. Lynch, and C. Newport. The abstract MAC layer. Technical Report MIT-CSAIL-TR-2010-040, CSAIL, MIT, 2010.

[16] F. Kuhn, N. Lynch, and C. Newport. The abstract MAC layer. *Distributed Computing*, 24(3):187–296, 2011.

[17] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In *Proc. of the 19th International Symposium on Distributed Computing*, pages 273–287, 2005.

[18] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[19] T. Moscibroda and R. Wattenhofer. Maximal independent sets in radio networks. In *Proc. of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 148–157, 2005.

[20] K. Nakano and S. Olariu. Randomized leader election protocols in radio networks with no collision detection. In *Proc. of the 11th International Conference on Algorithms and Computation*, pages 362–373, 2000.

[21] K. Nakano and S. Olariu. Leader election protocols for radio networks. In *Handbook of wireless networks and mobile computing*, pages 219–242. John Wiley & Sons, Inc., 2002.

[22] K. Nakano and S. Olariu. A survey of leader election protocols for radio networks. In *Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks*, pages 63–68, 2002.

[23] K. Nakano and S. Olariu. Uniform leader election protocols for radio networks. *IEEE Trans. on Parallel and Distributed Systems*, 13(5), 2002.

[24] J. Schneider and R. Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proc. of the 27th Symposium on the Principles of Distributed Computing*, pages 35–44, 2008.

[25] J. Schneider and R. Wattenhofer. What is the use of collision detection (in wireless networks)? In *Proc. of the 24th International Symposium on Distributed Computing*, pages 133–147, 2010.

[26] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7:585–600, 2001.

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