Self-stabilizing Dynamic Mutual Exclusion for Mobile Ad Hoc Networks

Yu Chen *, Jennifer L. Welch

Texas A&M University, College Station, TX 77843-3112, U.S.A.

Abstract

We propose a self-stabilizing mutual exclusion algorithm for mobile ad hoc networks, in which the composition of processors that want to enter the critical section can change dynamically. Our algorithm is based on dynamic virtual rings formed by circulating tokens. The algorithm always guarantees mutual exclusion and it guarantees different levels of progress under different levels of performance of the token circulation in the presence of mobility and message loss. Rigorous proofs of correctness and performance are given.

Key words: Self-stabilization, Mutual exclusion, Token circulation, Mobile ad hoc network

1 Introduction

Mobile ad hoc networks consist of mobile hosts, which are free to move arbitrarily. The communication between the mobile hosts depends on their positions and transmission ranges, so the communication topology may change with time as the hosts move into and go out of each other's transmission range. The technology of mobile ad hoc networking is becoming increasingly prevalent and it has been an active research area. But much of the work in this area has focused on routing and medium access control protocols, and there is less work on distributed services. Furthermore, past work on distributed services focused on non-fault-tolerant algorithms (e.g. leader election [9] [14], token circulation [13] [12] and mutual exclusion [1] [19]).

* Corresponding author. Phone (979)862-4535, Fax (979)847-8578
Email addresses: `ychen@cs.tamu.edu` (Yu Chen), `welch@cs.tamu.edu` (Jennifer L. Welch).

An important technique for designing algorithms that tolerate arbitrary transient faults is self-stabilization, first introduced by Dijkstra in [5]. A system is defined as self-stabilizing if, starting from an arbitrary configuration, it is guaranteed to converge to a "legitimate" configuration or a "legitimate" execution in finite time. The system in the legitimate configuration (or execution) exhibits desired behavior. The arbitrary starting configuration could be the result of a transient failure that corrupted some aspect of the system state. A lot of work has been done on self-stabilization for static networks. However, it is inefficient to apply most existing self-stabilizing algorithms for static networks directly in a mobile ad hoc network for the following reason. Traditionally, a topology change has been considered as a kind of transient fault. But most known self-stabilizing algorithms require no subsequent failures (i.e., static topology) in order to converge to a legitimate configuration. If such an algorithm is applied in a mobile ad hoc network, which typically experiences frequent topology changes, it would essentially start over in trying to converge every time a topology change occurs, which is quite inefficient and might cause the algorithm never to reach a legitimate configuration. Thus a self-stabilizing algorithm for mobile ad hoc network needs to be designed specifically so that it exhibits correct behavior in the presence of topology changes, once it has converged.

In this paper, we are interested in self-stabilizing distributed services for mobile ad hoc networks. We focus on the mutual exclusion problem. Informally, a mutual exclusion algorithm should satisfy two properties: *mutual exclusion* — there is no more than one processor in the critical section; and *progress* — some or all processors enter the critical section infinitely often. Our algorithm also provides a dynamic membership service so that processors can alternate participating in and not participating in the competition to enter the critical section. Mutual exclusion provides a mechanism for shared access to a resource, which is worth considering for mobile ad hoc networks, where mobile nodes may need to share common resources. Mutual exclusion can be used in the implementation of group communication, which is an important building block for applications that involve groups of cooperating hosts, such as mobile conferences, rescue missions and battlefield operations. Mutual exclusion is useful in many high level applications in mobile ad hoc networks such as E-learning in [11] to guarantee the consistency of shared objects.

Mutual exclusion and progress conditions cannot both be guaranteed in mobile ad hoc networks with arbitrary mobility for the following reason. Since reliable message delivery cannot be guaranteed in such a network, it is possible that some processors stop receiving messages from others even when they are always in the same connected component. If a processor makes the decision whether to enter the critical section based on communication with others, the progress condition cannot be guaranteed; if a processor can enter the critical section without communicating with others, mutual exclusion may be violated. Arbitrary mobility has prevented most protocols from providing fully reliable service in mobile ad hoc networks, and usually they provide a best-effort service instead. Our strategy to solve the mutual exclusion problem for mobile ad hoc networks is to guarantee mutual exclusion without constraints on the reliability of message delivery, and guarantee different levels of progress under different levels of constraints on mobility.

Our algorithm is based on a token circulation algorithm called LRV [13] [12], in which the token is forwarded by the current token holder to its neighboring processor that was visited by the token least recently. A distinguished processor in the system periodically generates a token. Upon receipt of a token, a processor checks the token state and its local state to decide whether it can enter the critical section; the check must ensure that only one processor is in the critical section even when there are multiple tokens in the system. Thus mutual exclusion can always be guaranteed, while progress depends on the frequency with which processors receive tokens.

2 Related Work

Mutual exclusion can be solved by circulating a token throughout the system in an appropriate way; when a processor has the token, it can enter the critical section. If the token circulation algorithm satisfies analogous properties (no more than one token exists in the system and every processor gets it infinitely often), clearly this simple approach to solving mutual exclusion is correct. Self-stabilizing mutual exclusion [5] and token circulation algorithms [4][15] have been designed for static networks. However, after they converge, they do not guarantee mutual exclusion in the presence of ongoing topology changes.

One paper on self-stabilization for mobile ad hoc networks is [8], in which a self-stabilizing group communication using random walk is proposed. In contrast to the *probabilistic* protocol in [8], the algorithm presented in our paper is *deterministic*.

Both algorithms use tokens or agents to control processors' behavior, but they differ in many aspects, including how the tokens or agents are forwarded, how membership is maintained and how to handle mobility. Other work on self-stabilization for mobile situations includes a "weakly" self-stabilizing leader election algorithm in [18]. Self-stabilization can be used to handle the topology changes in dynamic distributed system ([16], [7], [6]). A comprehensive bibliography on self-stabilization is given in [10]. A survey of fault-tolerant algorithms for mobile ad hoc networks can be found in [2].

A preliminary version of our algorithm appears in [3]. Simultaneously and independently, [1] presented a non-fault-tolerant mutual exclusion algorithm in which the ring is also built dynamically according to the current physical proximity of nodes. In this paper, we relax the requirement from [3] of static topology during stabilization, and extend the algorithm to handle dynamic membership, that is, the set of processors that want to enter the critical section can change.

3 Motivation

As mentioned above, a simple solution for mutual exclusion is to circulate a token throughout the network and only the processor which holds the token enters the critical section. If there is exactly one token in the system and this token visits every processor that wants to enter the critical section infinitely often, this simple idea solves the mutual exclusion task. But it is not self-stabilizing because it cannot recover from a configuration with zero or multiple tokens.

One idea to make this simple idea self-stabilizing is as follows. In order to recover from a configuration with no token, a distinguished processor keeps generating tokens, which are circulated throughout the network. Upon receipt of a token, a processor checks the token state and its local state to decide whether it can enter the critical section. Such check should ensure that eventually only one processor is in the critical section even when there are multiple tokens in the system. An example is the self-stabilizing mutual exclusion algorithm on a ring in the shared memory model presented in [5]. A message passing version of this algorithm is presented in [17], which works on a fixed ring with FIFO links. In this algorithm, each token and each processor has a token id. The distinguished processor, p_0, keeps generating tokens, setting the tokens' id as its token id. The tokens are forwarded

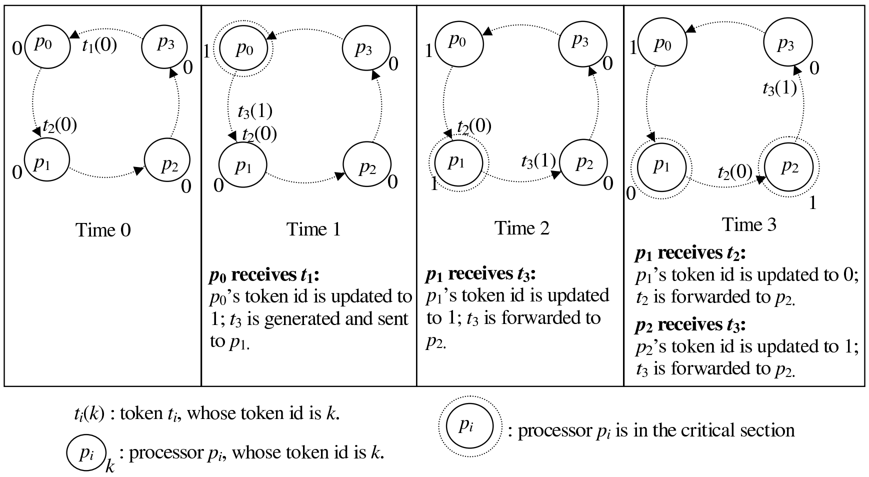

Fig. 1. A counter-example to the algorithm of [17] on a virtual ring with non-FIFO virtual links

along the ring. When processor p_i, $i \neq 0$, receives a token with id not equal to p_i's token id, p_i sets its token id to the token's id and enters the critical section. When a token comes back to p_0 after traveling around the whole ring, p_0 checks the token's id and its token id. If the token's id is equal to its token id, p_0 increases its token id by one and enters the critical section.

In this paper, we consider mutual execution in mobile ad hoc networks. If the composition of the network is fixed and known, we can predefine a *virtual ring*. Each token is circulated in the network by some token circulation algorithm, but it visits the processors according to their positions in the virtual ring: the first processor in a token's virtual ring is said to be visited by this token when it receives this token for the first time or the last processor visited by this token is the last processor in the virtual ring; each of the other processors p_i is said to be visited by a token if and only if the last processor visited by this token is the processor previous to p_i in the virtual ring. We notice the virtual link between two successive processors on the virtual ring is non-FIFO because of the changing topology — different tokens may travel along different paths from one processor to another. Applying the idea in [17] on such a virtual ring to solve mutual exclusion does not work with non-FIFO virtual links. A counter-example is shown in Fig. 1, in which the system has converged at time 0. But at time 3, processor p_1 and p_2 enter the critical section at the same time.

So we cannot apply the idea in [17] to mobile ad hoc networks directly. What is more, a fixed virtual ring is quite inefficient in mobile ad hoc networks as it may bear little resemblance to the actual topology. Finally, in a mobile ad hoc network, a processor p_i may be prevented from receiving a token due to an adversarial mobility pattern, and since all the processors that follow p_i in a virtual ring should be visited after p_i is visited, they will not be visited by any token and cannot enter the critical section any more, even if they keep receiving tokens.

In this paper we propose an algorithm based on the ideas in [5] and [17]. In order to handle the mobility, we need to deal with:

- dynamic virtual ring, which is updated periodically to reflect the changing topology,
- non-FIFO virtual links on the virtual ring, and
- partitions, so that a disconnected processor will not prevent others from entering the critical section.

4 System Model

In this paper a mobile ad hoc network is represented by a directed graph, in which each vertex v_i represents a mobile processor p_i and there is a directed link from vertex v_i to vertex v_j if and only if mobile processor p_j is within the transmission range of mobile processor p_i. We use the message passing model, in which each processor has an incoming queue. Messages in transit are modeled as being in the incoming queues of the receiving processors.

We focus on the distributed services that receive and respond to the requests from the application. The distributed services run on top of the basic communication service between adjacent processors. We make the following assumptions:

- $Assum_0$: There is a distinguished processor p_0 (we will discuss relaxing this assumption in section 9).
- $Assum_1$: An upper bound N on the number of processors is known, and each processor has a unique id.

  For simplicity, we assume processors' ids are in $[0, N-1]$ and processor p_i's id is i; the correctness of the algorithm does not rely on it. Note that we do not require the processors know the ids of others in advance.
- $Assum_2$: There is an upper bound, d, on the message delay between two adjacent

processors if the message is forwarded successfully. Messages may be lost but not corrupted.

We do not assume the network is always connected. Later we will see that message loss and the connectivity of the network will only affect the progress of the algorithm (see section 8.6) but not mutual exclusion. The assumption of no message corruption is required because otherwise the mutual exclusion problem cannot be solved even in a non-self-stabilizing way.

- $Assum_3$: There is an upper bound, g, on the number of messages generated by a processor in each time unit.

A transient fault may cause the system to enter an arbitrarily corrupted configuration. The property of self-stabilization models the ability of a system to recover from a corrupted configuration under the assumption that the transient faults do not continue to occur. Our algorithm can recover from an arbitrary configuration, which may be caused by any transient fault, under the above assumptions.

A self-stabilizing algorithm requires variables to be bounded since memory space is finite. In a non-self-stabilizing algorithm, an unbounded variable can be simulated by using large but finite memory: since the algorithm starts from a correctly initialized state, the maximum value that would ever be assigned to the variable in, say, the next few hundred years can be estimated and sufficient memory to store that value can be allocated. But this approach does not work for self-stabilizing algorithms, since in an arbitrary configuration, the variable could take on its maximum value, no matter how much space is allocated.

$Assum_1$ is useful to set bounds on the size of variables related to processors' identification. $Assum_2$ and $Assum_3$ enable the algorithm to rely on a maximum number of messages existing in the system, thus we can assign ids to messages from a bounded range.

There are timers on each processor such that each timer has a value which is decremented at the rate of real time, and if a timer is set for time interval T at time t, then at time $t + T$ an event is initiated by the timeout expiring.

Given a mobile ad hoc network, we refer to the state of a processor p_i, excluding the incoming queue, as "the local state" of p_i, denoted by $pstate_i$, which consists of the values of the local variables at p_i. We denote the incoming queue at p_i by $queue_i$, and the state of $p_i.queue$ by $qstate_i$, which consists of the sequence of messages in

$queue_i$. Thus the *configuration* can be described as

$$\{\langle pstate_0, \ldots, pstate_{n-1}\rangle, \langle qstate_0, \ldots, qstate_{n-1}\rangle, \tau\},$$

where τ is the directed graph representing the topology of the network.

In this paper, we study the execution of an algorithm in a mobile ad hoc network, in which topology changes may occur. There are two kinds of *events*: *local computation events* and *topology changes*. A local computation event is initiated by the receipt of a message, the receipt of a request from the application, or a timeout expiring. A local computation event changes the state of the processor at which it occurs, and enqueues messages in the incoming queues of neighboring processors. A topology change event changes the topology τ. A *timed event* is a pair consisting of an event and and a real number, which represents the real time at which the event occurs. We have the following assumption on the local computation.

- $Assum_4$: At each processor no more than one event is activated at the same time and the time of the local computation is negligible.

  Note that one consequence of this assumption is that the time a processor stays in the critical section is negligible. This assumption is made for simplicity. We will discuss relaxing it in section 9.

An *execution* of an algorithm in G is an alternating sequence of configurations and timed events $c_0, e_1, c_1, e_2, \ldots$ satisfying the following: (1) The real times of the events form a monotonically increasing sequence; if the execution is infinite, then the times increase without bound. (2) Messages are generated, sent and received in ways that are consistent with the assumptions listed above. (3) Each configuration after the first follows appropriately from the previous, according to the specification of the intervening event.

5 Problem Definition

We define the *mutual exclusion task* similarly to [5], in which "processor p_i being in the critical section in configuration c" refers to a predicate whose specifics will depend on the particular algorithm.

In some situations, processors may want to alternate between participating in and not participating in the competition to enter the critical section. In this paper, we consider a system in which the composition of the competing processors is changing as indicated by the processors receiving $join$ or $leave$ requests from the application. We call a processor in an execution *active* (*inactive* resp.) if it experiences only a finite number of $join$ and $leave$ requests and the last one is a $join$ ($leave$) request. Three versions of the *dynamic mutual exclusion task*, corresponding to different progress conditions, are defined as follows:

Definition 1 ***(Dynamic No Deadlock Mutual Exclusion.)*** *The task of dynamic no deadlock mutual exclusion is a set of executions, such that for every execution in the set,* $Mutual\ Exclusion$*,* $Eventual\ Stopping$ *and* $No\ Deadlock$ *are satisfied:*

- $Mutual\ Exclusion$*: There is no more than one processor in the critical section in any configuration.*
- $Eventual\ Stopping$*: Every inactive processor enters the critical section only finitely often.*
- $No\ Deadlock$*: Some active processor enters the critical section infinitely often if the execution is infinite.*

Variations with increasingly strong progress conditions are:

- ***Dynamic No Lockout Mutual Exclusion****, in which* $No\ Deadlock$ *is replaced with*
  - $No\ Lockout$*: Every active processor enters the critical section infinitely often if the execution is infinite.*
- ***Dynamic Bounded Waiting Mutual Exclusion****, in which* $No\ Deadlock$ *is replaced with*
  - $Bounded\ Waiting$*: There exists time interval* T*, such that every active processor enters the critical section exactly once in every time interval* T*.*

This paper describes an algorithm for mobile ad hoc networks that is self-stabilizing for $Mutual\ Exclusion$ and $Eventual\ Stopping$, and that is self-stabilizing for $No\ Deadlock$, $No\ Lockout$, and $Bounded\ Waiting$ with respect to various conditions (which are detailed later).

Definition 2 **(Self-Stabilization).** *An algorithm is self-stabilizing for property* P *(with respect to condition* C*), if, starting from an arbitrary configuration, every* *infinite execution of the algorithm (satisfying C) has a suffix satisfying P.*

In this paper, property P can be one of $No\ Deadlock$, $No\ Lockout$ and $Bounded\ Waiting$, and condition C captures the assumptions about the network behavior.

6 Algorithm Overview

In this section, we give an informal description of our algorithm. Communication in our algorithm is based on token circulation, where each token is a message. We have two kinds of tokens: *mutual exclusion token* and $join\_request$ *token*, denoted by m-$token$ and j-$token$ respectively; the specific usage of each kind of token is described later in this section. Both kinds of tokens are routed using a self-stabilizing version of a token circulation algorithm called LRV from [13], [12]. The reason for using LRV (instead of a more traditional routing algorithm) is that we have developed a simple self-stabilizing version of LRV, whereas it is much more problematic to design a self-stabilizing version of traditional routing algorithms. Our version of LRV uses bounded timestamps and enforces a lifetime on tokens, by discarding any token that has been forwarded more than a certain number of hops. Because of the bounded lifetime, we can assign ids to tokens from a bounded range and we define the operations on token ids to be the operations on integers mod this bound. Note in section 4, we explained why the values of a variable should be from a bounded range.

The execution of our algorithm is divided into phases. Informally, a phase is a bounded subsequence of execution in which all the connected members enter the critical section exactly once. The special processor p_0 maintains the membership, which is organized in two variables: new_set, a set of processors that are waiting to join the system, and $ring$, a list of members that have joined the system. The membership is updated only at the beginning of each phase.

- In each phase, p_0 repeatedly generates m-$token$s which carry new_set and $ring$.
  - Processors in new_set initialize their local states upon receipt of an m-$token$.
  - Members in $ring$ are visited by an m-$token$ according to their positions in $ring$. When a member is visited by an m-$token$,
    - it checks the token's state and its local state to see whether it is in the critical section, and updates its local state accordingly (the condition and

update is designed to handle non-FIFO virtual links, see Section 7 for details);
    * if it wishes to leave the system, it indicates this in the $m\text{-}token$.
  - Each $m\text{-}token$ records the order of processors that have forwarded it.
- When a processor wishes to join the system, it periodically sends a $j\text{-}token$ to processor p_0. The sending is repeated in order to increase the likelihood that the message gets through even if some copies are lost due to mobility.
- Processor p_0 starts a new phase when all the processors in new_set have initialized their states and all the members in $ring$ have entered the critical section, or a certain amount of time has elapsed since the current phase began. A member is considered to be disconnected if it did not enter the critical section in the previous phase. The member list is updated by p_0 as follows:
  - $ring$ is updated to be the currently reachable members, which are the processors in new_set that have initialized their local state and the processors in $ring$ that have entered the critical section in last phase, minus the leave requests. The order of members in $ring$ is decided by the order of processors that have forwarded the $m\text{-}token$s. Since a poorly chosen virtual ring requires a long time for a token to visit all the nodes, the idea behind this heuristic is to have the visiting order match more closely the actual network topology in order to increase efficiency.
  - new_set is updated to be the senders of $j\text{-}token$s received by p_0 in the last phase.

7 Algorithm

In this section, we introduce the underlying token circulation algorithm in section 7.1, the data structures in section 7.2, the interfaces to the application in section 7.3, and the mutual exclusion algorithm in section 7.4.

7.1 Token circulation algorithm: LRV_L

The non-self-stabilizing LRV algorithm is introduced in [13], [12], in which there is a unique token t that contains an array ts of N integers (the "*timestamp array*") initialized to all 0's; $ts[i]$ is a $timestamp$ indicating when processor p_i received the token most recently. In LRV, a token is forwarded to a neighboring processor that was visited by this token least recently.

In our algorithm we use a variant of LRV, denoted by LRV_L, which is similar to LRV except that the timestamps in LRV_L are in the range $[0, L]$, the operation "+" on elements of $t.ts$ is addition mod $(L + 1)$, and only tokens with maximum timestamp less than L are forwarded to the next processor (see Fig. 2.) The value of L should be chosen based on the expected network behavior to guarantee that most of the tokens can be forwarded to every processor within L hops (see section 8.6 for further discussion). If token t is not routed to the next processor, we say t is no longer in the system. In the sequel, we refer to t's *lifetime* as the time token t has stayed in the system. In this paper, we consider a self-stabilizing algorithm, which may start from an arbitrary configuration. The following lemma shows that the lifetime of every token forwarded by LRV_L is bounded no matter what values are initially in the timestamp array.

Lemma 1 *Consider any execution σ, in which tokens are forwarded for LRV_L. The lifetime of any token in σ is no more than $M_{lt} = L \cdot d$. In more detail, in any configuration c, token t has been in the system for no more than $max(t.ts) \cdot d$, and t can be in the system for no more than $(L - max(t.ts)) \cdot d$ hops after c.*

Proof. Denote the initial timestamp array of token t by $t.ts^0$. From the code we can see $max(t.ts)$ is increased by 1 each time t is forwarded, so t cannot be forwarded for more than L hops, and t's lifetime is no more than $L \cdot d$ by $Assum_2$. Suppose in configuration c, t is in p_i's incoming queue. Then t has been forwarded for $max(t.ts) - max(t.ts^0)$ hops, and it cannot be forwarded for more than $L - max(t.ts)$ hops after c. By $Assum_2$, t has stayed in the system for no more than $max(t.ts) \cdot d$ time, and t can stay in the system for no more than $(L - max(t.ts)) \cdot d$ after c time. ∎

$LRV_L\_initialize(token\ t)$:
$t.ts = [0, 0, \cdots, 0]$;

$LRV_L\_forward(token\ t)$:
$t.ts[i] = max(t.ts) + 1$;
if $(max(t.ts) \geq L)$ **then return**;
$next$ = index of $t.ts$ entry among all neighbors
    of p_i with minimum value (break ties by id);
send t to p_{next};

Fig. 2. Token circulation algorithm: LRV_L

7.2 Data Structures

The denotations in Fig. 3 are used in our algorithm, where L is the upper bound on the number of hops taken by tokens that is enforced by LRV_L. The choice of L only affects the progress of the algorithm (see section 8.6), but not mutual exclusion. Note M_{tidr} is the maximum difference between any two token ids at any given time after convergence, while M_{tid} is the bound on the $token\_id$ type. The reset values of each timer are listed in Fig. 4 (the explanation of each timer is listed in Fig. 5.) Recall that N, g and d were introduced in section 4.

Denotation	Value	Explanation
M_{lt}	$L \cdot d$	upper bound on token's *lifetime*
M_{tidr}	$(3g + 1)M_{lt} + 2$	maximum difference between any two token ids after convergence, i.e. range of token ids (difference is defined in this section)
M_{tid}	$2M_{tidr} + 2$	upper bound on $token\_id$ variables

Fig. 3. Denotations

$Timers$		Reset values	Requirement on reset values
$timers$ on p_0	$timer_0^{phase}$	TM_{phase}	$> M_{lt}$
	$timer_0^{gen}$	TM_{gen}	
$timers$ on p_i, $i \neq 0$	$timer_i^{app}$	TM_{app}	$> (3M_{lt} + 3TM_{phase})$
	$timer_i^{join}$	TM_{join}	$> (M_{lt} + 3TM_{phase})$
		TM_{join_re}	$< TM_{join}$

Fig. 4. Timers' reset values

We define the $token\_id$ type to be the set of integers in $\{0, 1, \ldots, M_{tid}\}$. The operations on variables of type $token\_id$ variables are the operations on integers mod $(M_{tid} + 1)$. In particular, given two $token\_id$ variables, tid_1 and tid_2, the operation $[tid_1, tid_2]$ returns $\{tid_1, tid_1 + 1, \ldots, tid_2\}$ if $tid_1 \leq tid_2$, and $\{tid_1, tid_1 + 1, \ldots, M_{tid}\} \cup \{0, 1, \ldots, tid_2\}$ otherwise. We define the *difference* between tid_1 and tid_2 as $min\{t_{max} - t_{min}, t_{min} - t_{max} + M_{tid} + 1\}$, where $t_{max} = max\{tid_1, tid_2\}$ and $t_{min} = min\{tid_1, tid_2\}$. Note that the difference between tid_1 and tid_2 is the same as the difference between tid_2 and tid_1. For example, if $M_{tid} = 12$, then $[5, 9] = \{5, 6, 7, 8, 9\}$, $[9, 5] = \{9, 10, 11, 12, 0, 1, 2, 3, 4, 5\}$, the difference between 5 and 9 is 4 and the difference between 2 and 9 is 5.

The fields on tokens and the variables at processors are listed in Fig. 5.

Token	Field	Data Type	Explanation
j-$token$	ts	$timestamp$ array	used by LRV_L
	pid	integer in $[0, N]$	id of sender
m-$token$	ts	$timestamps$ array	used by LRV_L
	id	$token\_id$	token id
	$ring$	permutation of subset of $[0, N]$	processors that have joined
	new_set	subset of $[0, N]$	joining processors
	$visit\_set$	subset of $[0, N]$	processors visited
	$route\_list$	permutation of subset of $[0, N]$	the order in which the token was routed
	$leave\_set$	subset of $[0, N]$	leave requests

Processor	Variable	Data Type	Explanation
p_0	tid_0	$token\_id$	p_0's local token id
	$ring_0$	permutation of subset of $[0, N]$	processors that have joined
	new_set_0	subset of $[0, N]$	joining processors
	$visit\_set_0$	subset of $[0, N]$	processors visited
	$route\_list_0$	permutation of subset of $[0, N]$	the order in which the token was routed
	$join\_set_0$	subset of $[0, N]$	join requests
	$leave\_set_0$	subset of $[0, N]$	leave requests
	$discon_0$	boolean	indicates whether there exist partitions
	$timer_0^{gen}$	$timer$	for token generation
	$timer_0^{phase}$	$timer$	for starting a new phase
p_i,	tid_i	$token\_id$	p_i's local token id
$i \neq 0$	$member_i$	boolean	indicates whether p_i is a member
	$status_i$	$joined/joining$	indicates whether p_i has joined
	$timer_i^{join}$	timer	for resending j-$tokens$
	$timer_i^{app}$	timer	time between application requests

Fig. 5. Fields on tokens and variables at processors

7.3 Interfaces to the Application

The application process on processor p_i submits a $join$ or $leave$ request to the mutual exclusion service by setting $member_i$ to $true$ or $false$ (see events $join_i$ and $leave_i$ in section 7.5.) A processor cannot join the system if it has currently joined, that is, $member_i = true$ (see precondition of $join_i$ in section 7.5), and it cannot leave if it is not in the system, that is, $member_i = false$ (see precondition of $leave_i$ in section 7.5). The time interval between two requests is no less than TM_{app}, which is enforced by using $timer_i^{app}$ (see line 2.2 of $leave_i$ and the precondition of $join_i$ in section 7.5). Such an interval is required because when a processor leaves the system, the system needs time to erase the information related to it, so that when it rejoins the system, it will not update its state based on false information.

The application decides whether it can enter the critical section based on the part of the configuration accessible to p_i, that is, p_i's local state and the state of its incoming queue (see Predicate cs_i for p_i to enter the critical section, section 7.5.) Processor p_0 enters the critical section at the beginning of each phase (see cs_0 and the precondition of event $start\_phase_0$). Processor p_i, $i \neq 0$, enters the critical section if p_i has joined the system (see cs_i^1), it is p_i's turn to be visited (see cs_i^2), and p_i has a correct token id (see cs_i^3). We notice in a $receive_i^m$ event following any configuration in which p_i is in the critical section, line 9.2 is $false$, line 9.8 is $true$, and line 9.13 is $true$.

7.4 Mutual Exclusion Algorithm

Mutual exclusion is achieved by four kinds of events on processor p_0 and three kinds of events on the other processors (see code in section 7.5). Processor p_0 keeps generating $m\text{-}token$s in event $generate_0^m$. A new phase is started by p_0 in event $start\_phase_0$. Event $receive_0^j$ is activated when p_0 receives a $j\text{-}token$, and event $receive_0^m$ is activated when it receives an $m\text{-}token$. Processor p_i, $\neq 0$, sends j-$token$s to p_0 in event $send_i^j$, which are forwarded to p_0 among processors in event $receive_i^j$. Event $receive_i^m$ is activated when p_i receives an $m\text{-}token$.

Now we explain how these events interact with each other to achieve mutual exclusion. Processors join the system by sending join requests. Upon receipt of a $join$ request from the application, p_i sets $status_i$ to $joining$, indicating it is waiting to join the system, and sends $j\text{-}token$s (lines 7.1 - 7.6), which are forwarded to p_0 by processors using LRV_L (lines 8.1 - 8.2). Upon receipt of such a token (line 6.1 - 6.4), p_0 records the information in its local variable $join\_set_0$, which will be used to update the membership at the begining of the next phase. Note in event $send_i^j$, a processor resends the join request in every TM_{join_re} time interval if it does not receive an $m\text{-}token$ within time TM_{join} by using the timers.

Now we consider the execution of each phase. In each phase, p_0 repeatedly generates $m\text{-}token$s (lines 3.1 - 3.10). Each $m\text{-}token$ has an id and carries membership information in fields new_set and $ring$. Note that since p_0 updates its local token id and membership only at the beginning of each phase, all the tokens generated in the same phase have the same token id and membership information.

We now explain how a processor p_i, $i \neq 0$, responds to the receipt of an m-$token$ t (lines 9.1 - 9.24).

- If p_i is waiting to join the system and its join request was received by p_0 in the last phase, line 9.2 is true when p_i receives the first m-$token$ that is generated in the current phase. In this case, p_i takes following actions (line 9.3-9.6): it initializes its local token id tid_i as $t.id$ and sets $status_i$ to $joined$, indicating it has initialized its local state and joined the system; it also adds its id to $t.visit\_set$ (line 9.5), ensuring that it will ignore token t if it receives t again. (As we explain below, once p_i joins the system, p_i will be moved from new_set to $ring$ in the next phase, and line 9.2 will be false in the next phase.)
- Otherwise, we say the processor is "visited" by m-$token$ t if line 9.8 is true. Note that this condition is true only if p_i is a member in $ring$ and it enforces the property that each token visits processors according to their positions in the $ring$. In line 9.9 p_i adds its id to $t.visit\_set$, ensuring that it will ignore token t if it receives t again. Processor p_i updates its local state and checks whether it is in the critical section. If $t.id$ is 0 and p_i's token id is currently out of the range $\{M_{tid}, 0, \ldots, M_{tidr}\}$, then an inconsistency in the state of the system has been detected, and p_i's token id is reset to 0 (lines 9.10 - 9.12). If $t.id$ is one more than p_i's token id, then p_i enters the critical section (lines 9.13 - 9.15). After leaving the critical section, p_i sets its token id to be $t.id$ (line 9.14), which will prevent p_i from entering the critical section again due to tokens generated in the current phase.

  The intuition of how it works follows. Formal proofs are provided in Section 8.
  - Consider some phase b in which p_0 keeps generating m-$token$s with id 0. All the visited processors have local token id in range $\{M_{tid}, 0, \ldots, M_{tidr}\}$ at line 9.12 and those that have token id M_{tid} at line 9.12 will enter the critical section and update their token id to 0 in line 9.14 since $t.id = 0$. Thus all the visited processors will have token ids in $[0, M_{tidr}]$ at the end of phase b.
  - In phase $b + 1$, p_0 keeps generating token with id 1 and all the processors with token id 0 update their token id to 1 when visited by such a token. Processors with other token ids will not change their token id since the bounded token lifetime guarantees that no token with id in $[2, M_{tidr} + 1]$ exists in phase $b + 1$, and only tokens with id in this range can change a processor's token id that is in $[1, M_{tidr}]$ (see lines 9.10 and 9.13). So at the end of phase $b + 1$, all the

visited processors have id in $[1, M_{tidr}]$.

· Generally speaking, at the begining of phase $b + k$, $1 \leq k \leq M_{tidr}$, all the processors have token ids in $[k-1, M_{tidr}]$. In phase $b+k$, p_0 keeps generating tokens with id k. All the processors with id $k-1$ updates their token ids to k when visited by a token generated in the current phase, while others remain unchanged since no token with id in range $[k+1, M_{tidr}+1$ exists due to the bounded token lifetime. Thus at the end of phase $b+k$, all the visited processors have token ids in $[k, M_{tidr}]$. So all the visited processors have the same token id at the end of phase $b + M_{tidr}$; we say the system converges at this point.

· After convergence, at the begining of phase $b + k$, $k > M_{tidr}$, all the members have token id $k-1$. In this phase p_0 keeps generating tokens with id k, and the members enter the critical section only when they are visited by a token generated in the current phase for the first time. Since all these tokens visit processors in the same order, specified by $ring$, $Mutual\ Exclusion$ is guaranteed, even if the virtual links are non-FIFO.

- When a processor wishes to leave the system, it indicates this in the $m\text{-}token$s that pass through it (line 9.20). The order of processors that have forwarded the token is updated in line 9.22. At last, the token is forwarded to the next processor using LRV_L (line 9.24).

When p_0 receives an $m\text{-}token$, it only processes the token if it has the same id as p_0's token id, since only these tokens can enable a processor to enter the critical section after the system converges (lines 5.1 - 5.11). The information about reachable members, routing order and leave requests are collected in line 5.3-5.5. If all the members (those in $ring$) have entered the critical section and all the waiting processors (those in new_set) have initialized their local state, that is, line 5.6 is true, then p_0 sets $timer_0^{phase}$ to 0 in line 5.7 to start a new phase (see the precondition of event $start\_phase_0()$). Otherwise the token is forwarded to the next processor in line 5.9.

A new phase is started in event $start\_phase_0$ by p_0 (lines 4.1 - 4.11). There are two situations for the precondition to be true. The first is line 5.7 as mentioned above. The other is where timer $timer_0^{phase}$ goes off; the purpose of $timer_0^{phase}$ is to prevent a phase from never ending due to partition. In this event, p_0 updates the membership: new_set consists of the processors whose join requests were received in the last phase (line 4.2), and $ring$ is the currently reachable members, (those in new_set that have initialized their local state and those in $ring$ that have entered the critical section), minus the leave requests (line 4.3) . Processor p_0 also figures out if the system has become disconnected, by checking whether any of the processors in the old group were not visited yet did not send a leave request (line 4.1). Since m-$token$s generated in the last phase continue circulating, a disconnected processor may be visited by an m-$token$ which enables it to enter the critical section in the new phase. In order to avoid the situation that a disconnected processor enters the critical section at the same time as a member enters, p_0 waits until all the m-$token$s are discarded to generate new tokens for the new phase (lines 4.6-4.9). Note that a disconnected member can rejoin the system by sending j-$token$s — as mentioned above, processor p_i resends j-$token$s in event $sent_i^j$ if it does not receive an m-$token$ within time TM_{join}.

7.5 Code

The predicate for p_i to enter the critical section and the code of each event on the processors are listed as follows:

Predicate cs_i for p_i to enter the critical section:

- $i = 0$: $cs_0 \equiv (timer_0^{phase} = 0)$
- $i \neq 0$: $cs_i \equiv cs_i^1 \wedge cs_i^2 \wedge cs_i^3$,
  where cs_i^1,cs_i^2 and cs_i^3 are defined as follows, in which m-$token$ t is the first token in $queue_i$.

$cs_i^1 \equiv (member_i = true) \wedge (status_i = joined)$

$cs_i^2 \equiv p_i$ is the first processor in $t.ring$ that is not in $t.visit\_set$

$cs_i^3 \equiv t.id = (tid_i + 1)\ mod\ (M_{tid} + 1)$

/* Event $join_i$ occurs when p_i receives a request from the application to join the system */
Event $join_i()$ on $p_i, i \neq 0$:
pre: Receipt of $join$ request from the application, and $(member_i = false) \wedge (timer_i^{app} = 0)$
action:
1.1 $member_i = true$;

/* Event $leave_i$ occurs when p_i wants to leave the system. p_i can rejoin only after time TM_{app} */
Event $leave_i()$ on $p_i, i \neq 0$:
pre: Receipt of $leave$ request from the application, and $(member_i = true)$
action:
2.1 $member_i = false$;
2.2 $timer_i^{app} = TM_{app}$;

/* p_0 continues generating m-$token$s in event $generate_0^m$ */
Event $generate_0^m()$ on p_0:
pre: $timer_0^{gen} = 0$
action:
3.1 create new m-$token$ data structure t;
3.2 $LRV_L\_initialize(t)$;
3.3 $t.id = tid_0$;
3.4 $t.ring = ring_0$;
3.5 $t.new\_set = new\_set_0$;
3.6 $t.visit\_set = \{p_0\}$;
3.7 $t.route\_list = \langle p_0 \rangle$;
3.8 $t.leave\_set = \emptyset$;
3.9 $LRV_L\_forward(t)$;
3.10 set $timer_0^{gen} = TM_{gen}$;

/* A new phase is started by p_0 in event $start\_phase_0$ */
Event: $start\_phase_0()$ on p_0:
pre: $timer_0^{phase} = 0$
action
/* Update membership */
4.1 $discon_0 = !((ring_0 - visit\_set_0) \subseteq leave\_set_0)$;
4.2 $new\_set_0 = join\_set_0$;
4.3 $ring_0$ = elements of $(visit\_set_0 - leave\_set_0)$ in the order in which they appear in $route\_list_0$ (if there exist elements in $(visit\_set_0 - leave\_set_0 - route\_list_0)$,

append them to the end);

/* Update token id and reset variables */
4.4 $tid_0 = (tid_0 + 1)\ mod\ (M_{tid} + 1)$;
4.5 $join\_set_0 = leave\_set_0 = \emptyset, visit\_set_0 = \{p_0\}$;

Critical Section

/* Clean up tokens if there are disconnected members: $timer_0^{gen}$ is set to M_{lt} to ensure p_0 stops generating tokens for M_{lt} time */
4.6 **if** $(discon_0 == true)$ **then**
4.7 set $timer_0^{gen} = M_{lt}$;
4.8 **else**
4.9 set $timer_0^{gen} = 0$;
4.10 **endif**

/* Reset phase timer */
4.11 set $timer_0^{phase} = TM_{phase}$;

/* Event $receive_0^m$ occurs when p_0 receives a $m\text{-}token$. */
Event $receive_0^m(m\text{-}token\ t)$ **on** p_0**:**
pre: The first token in $queue_0$ is $m\text{-}token\ t$.
action:
5.1 remove t from $queue_0$;

5.2 **if** $(t.id == tid_0)$ **then**
/* Update membership information */
5.3 $visit\_set_0 = visit\_set_0 \cup t.visit\_set$;
5.4 $route\_list_0 = t.route\_list$;
5.5 $leave\_set_0 = leave\_set_0 \cup (t.leave\_set \cap ring_0)$;

/* If the token has visited all the members, start a new phase, otherwise forward the token to the next processor using LRV_L*/
5.6 **if** $((ring_0 \cup new\_set_0) \subseteq visit\_set_0)$ **then**
5.7 set $timer_0^{phase} = 0$;
5.8 **else**
5.9 $LRV_L\text{-}forward(t)$;
5.10 **endif**
5.11 **endif**

/* Event $receive_0^j$ occurs when p_0 receives a join request. Membership is updated accordingly */
Event: $receive_0^j(j\text{-}token\ t_j)$ **on** p_0**:**
pre: The first token in $queue_0$ is $j\text{-}token\ t_j$.
action
6.1 remove t_j from $queue_0$;
6.2 **if** $(t_j.pid \notin (ring_0 \cup new\_set_0))$ **then**
6.3 $join\_set_0 = join\_set_0 \cup \{t_j.pid\}$;
6.4 **endif**

/* In event $send_i^j$, p_i continues generating $j\text{-}tokens$ to increase the likelihood that such request arrives at p_0 */
Event $send_i^j()$ **on** $p_i, i \neq 0$:
pre: $(member_i = true) \wedge (timer_i^{join} = 0)$
action:
7.1 create new $j\text{-}token$ data structure t_{join};
7.2 $LRV_L\text{-}initialize(t_{join})$;
7.3 $t_{join}.pid = p_i$;
7.4 $LRV_L\text{-}forward(t_{join})$;
7.5 $status_i = joining$;
7.6 set $timer_i^{join} = TM_{join\_re}$;

/* Event $receive_i^j$ occurs when p_i receives a join request. The join request is forwarded to p_0 using LRV_L */
Event $receive_i^j(j\text{-}token\ t_j)$ **on** $p_i, i \neq 0$:
pre: The first token in $queue_i$ is $j\text{-}token\ t_j$
action:
8.1 remove t_j from $queue_i$;
8.2 $LRV_L\text{-}forward(t_j)$;

/* Event $receive_i^m$ occurs when p_i receives a $m\text{-}token$. p_i checks its local state and the token's state to see whether it can enter the critical section */
Event $receive_i^m(m\text{-}token\ t)$ **on** $p_i, i \neq 0$:
pre: The first token in $queue_i$ is $m\text{-}token\ t$.
action:
9.1 remove t from $queue_i$;
9.2 **if**$((member_i == true)\&\&(status_i == joining)\&\&\ (p_i \in t.new\_set)\&\&(p_i \notin t.visit\_set))$ **then**
/* If p_i is waiting to join the system, it initializes its local state based on the token's state*/
9.3 $tid_i = t.id$;
9.4 $status_i = joined$;
9.5 $t.visit\_set = t.visit\_set \cup \{p_i\}$;
9.6 set $timer_i^{join} = TM_{join}$;
9.7 **else**
9.8 **if** $((member_i == true)\ \&\&\ (status_i == joined)\ \&\&\ (p_i$ is the first processor in $t.ring$ not in $t.visit\_set))$ **then**
/* If p_i is in the member list, it updates the token's state and its local state, and checks whether it can enter the critical section */
9.9 $t.visit\_set = t.visit\_set \cup \{p_i\}$;
/* Reset local token id if it is out of range */
9.10 **if** $(t.id == 0)\&\&(tid_i \notin \{M_{tid}, 0, \ldots, M_{tidr}\})$ **then**
9.11 $tid_i = 0$;
9.12 **endif**
/* Check whether it can enter critical section */
9.13 **if** $(t.id == (tid_i + 1)mod(M_{tid} + 1))$ **then**

Critical Section

9.14 $tid_i = t.id$;
9.15 **endif**
9.16 set $timer_i^{join} = TM_{join}$;
9.17 **endif**
9.18 **endif**

9.19 **if** $(member_i \neq true)$ **then**
/* Attach leave request if p_i is leaving */
9.20 append p_i to $t.leave\_set$;
9.21 **else if** $(p_i \notin t.route\_list)$ **then**
9.22 append p_i to $t.route\_list$;
9.23 **endif**
9.24 $LRV_L\text{-}forward(t)$;

8 Correctness

In this section, we show the correctness by proving that eventually tokens, non-members and members will all exhibit good behaviors. The organization of the proof is shown in Fig. 6. We explain in more detail below.

In section 8.1, we show $Eventual\ Stopping$ is true from the initial configuration of any execution (c_0 in Fig. 6), and introduce the term "phase". Informally, a phase is a bounded subsequence of an execution in which the membership and tid_0 remain unchanged (they are updated only at the beginning of each phase) and each connected member enters the critical section exactly once.

We discuss the properties of m-$token$s in section 8.2. In an arbitrary configuration, the ids of tokens may spread throughout the range $[0, M_{tid}]$. We show that eventually they will converge to a narrow range; this property is called $token\_safe$. We prove it by defining property $conv_{token}$ of a configuration and showing that eventually the execution will reach a configuration satisfying $conv_{token}$ (c_1 in Fig 6) and after that $token\_safe$ is always guaranteed.

The properties of non-members are discussed in section 8.3. In an arbitrary configuration, there may exist false information related to processors which are no longer members, based on which these processors may violate mutual exclusion when they join the system. We show that eventually a processor can join the system only when no information related to it exists; this property is called $nmem\_safe$. We prove it by first defining property $conv_{nmem}$, in which we define the appropriate levels of information in a configuration related to a non-member according to the time it has left the system. Then we show that any execution will eventually reach a configuration satisfying $conv_{nmem}$ (c_2 in Fig. 6)and after that $nmem\_safe$ is always guaranteed.

Then in section 8.4, we focus on the properties of members in an execution in which all the configurations are both $token\_safe$ and $nmem\_safe$. We define property $conv_{member}$ of a configuration related to members' state. We prove eventually the execution will reach a configuration satisfying $conv_{member}$ (c_3 in Fig. 6), and we say the execution has converged if it has reached such a point.

In section 8.5, we prove $Mutual\ Exclusion$ in any execution that has converged.

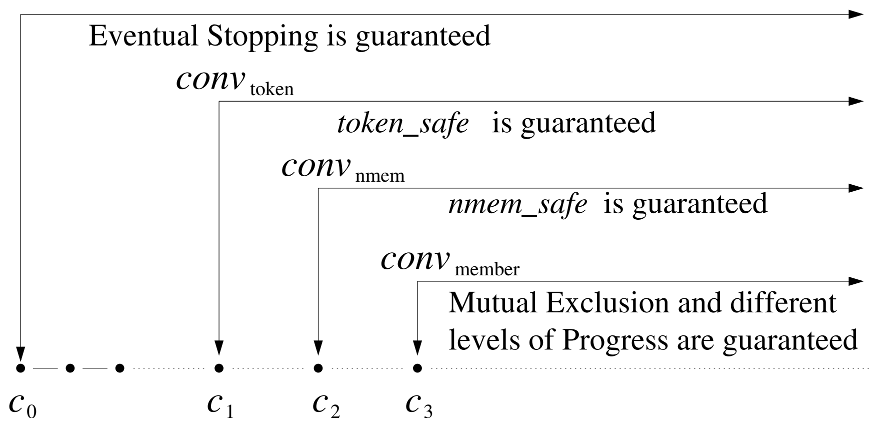

Fig. 6. Organization of proof

We also provide a discussion of different levels of progress achieved by the algorithm given different performance of token circulation in section 8.6.

8.1 Properties of Execution

Since $member_i$ is set to $false$ when p_i receives a $leave$ request from the application (see section 7.5), p_i cannot enter the critical section before the next $join$ request. Thus $Eventual\ Stopping$ can be guaranteed.

Theorem 2 *Given any execution σ, $Eventual\ Stopping$ is guaranteed in σ.*

Our algorithm is based on circulating m-$token$s, which are forwarded by LRV_L, but *visit* processors based on their member list. A formal definition of "visiting" is given as follows, from which we see that an m-$token$ t visits processors in $t.ring$ according to their positions in $t.ring$ (see line 9.8), and each processor in $t.ring$ or $t.new\_set$ is visited by t at most once (see line 9.5 and line 9.9).

Definition 3 ***(Visiting).*** *Given an m-$token$ t, if t is the first token in $queue_i$ in a configuration c, p_i is visited by t in c if and only if line 9.2 or line 9.8 is $true$.*

We divide the execution into "phases", which are defined as follows (see Fig. 7):

Definition 4 **(phase).** *Given an execution σ, σ is divided into phases:*

- *phase 0 starts at the beginning of σ, and ends just before the first $start\_phase_0$ event in σ; and*
- *phase i, $i > 0$, starts immediately after the end of phase $i-1$ and ends just*

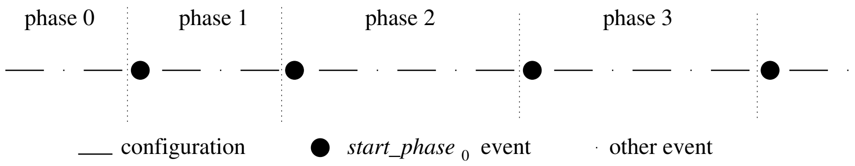

Fig. 7. Definition of phases

before the second occurrence after phase $i-1$ of a $start\_phase_0$ event.

In the sequel we call the $m\text{-}token$s with id equal to p_0's token id as "current" tokens. We denote the $start\_phase_0$ event of phase k by e_s^k, the configuration previous to e_s^k by c_s^k, the first $generate_0^m$ event of phase k by e_g^k and the configuration following e_g^k by c_g^k. We have the following properties for each phase:

Lemma 3 *Given any execution σ and any $k \geq 0$,*

(1) *In phase k, tid_0, $ring_0$ and new_set_0 remain unchanged; their values in phase k are denoted by tid_0^k, $ring_0^k$, and $new\_set_0^k$.*
(2) *All tokens generated in the same phase have the same values for id, $ring$ and new_set.*
(3) *The ids of the $m\text{-}tokens$ generated in a phase are one larger than those of the $m\text{-}tokens$ generated in the previous phase, that is, $tid_0^k = tid_0^{k-1} + 1$.*
(4) *The upper bound on the phase length is TM_{phase}.*

Proof. All $m\text{-}token$s generated by p_0 are generated with $tid = tid_0, ring = ring_0$ and $new\_set = new\_set_0$ in the $generate_0^m$ event. Since the only update to $ring_0$, new_set_0 and tid_0 is line 4.1-4.4 in a $start\_phase_0$ event, in which tid_0 is increased by one, (1), (2) and (3) are proved.

A $start\_phase_0$ event is activated if and only if $timer_i^{phase} = 0$, in which $timer_i^{phase}$ is set to TM_{phase}. After that, $timer_i^{phase}$ keeps going down until it goes off or it is set to 0 by line 5.7 in a $receive_0^m$ event, so it takes at most TM_{phase} for a $start\_phase_0$ event to be activated again. Thus (4) is proved. ■

8.2 Safe Properties of $m\text{-}tokens$

In this section, we discuss the properties related to $m\text{-}token$s. In an initial configuration, the id of a token may be any one of $[0, M_{tid}]$. We show in Lemmas 5 and 6 that eventually the ids of tokens will converge to some subset of $[0, M_{tid}]$ (property

$conv_{token}$) and starting from that point, there always exists an upper bound M_{tidr} on the difference of token ids in a configuration (property $token\_safe$.) An execution in which all the configurations satisfy this bound exhibits good behaviors: a processor's token id remains unchanged when certain conditions hold (Lemma 7), all the current tokens are generated in the current phase (Lemma 8), and all the members are those visited by a current token in the previous phase (Lemma 9).

First we define properties $conv_{token}$ and $token\_safe$.

Definition 5 *($conv_{token}$). A configuration c satisfies property $conv_{token}$ if and only if*

(1) *the number of m-$tokens$ is no more than $g \cdot M_{lt}$;*
(2) *for every m-$token$ t, we have $t.id \in [tid_0 - (g \cdot M_{lt} + g \cdot d \cdot max(t.ts) + 1)\ ,\ tid_0]$;*
(3) *any two m-$tokens$ with the same token id have the same $ring$ and new_set;*
(4) *any m-$token$ t with id tid_0 satisfies $(t.ring = ring_0) \wedge (t.new\_set = new\_set_0)$.*

Definition 6 *($token\_safe$). A configuration is $token\_safe$ if and only if:*

(1) *for any m-$token$ t, we have $t.id \in [\ tid_0 - M_{tidr}\ ,\ tid_0]$;*
(2) *all the m-$tokens$ with the same id have the same $ring$ and new_set.*

We will show in Lemmas 5 and 6 that, starting from an arbitrary configuration, eventually the execution converges to a configuration satisfying $conv_{token}$, after which $token\_safe$ is always guaranteed. The proof of this lemma is based on the fact that the number of phases is limited in each bounded time interval.

Lemma 4 *Given times T_1 and T_2, $T_1 < T_2$, if the number of m-$tokens$ at T_1 is no more than k, then the number of $start\_phase_0$ events in $[T_1, T_2]$ is no more than ($k+\ g \cdot (T_2 - T_1) + \left\lceil \frac{T_2 - T_1}{TM_{phase}} \right\rceil$).*

Proof. A $start\_phase_0$ event is activated only when $timer_0^{phase} = 0$, which occurs only in two situations:

- $timer_0^{phase}$ is set to 0 in a $receive_0^m$ event. The number of such events is no more than the number n_r of m-$token$s received by p_0 in $[T_1, T_2]$, which either exist at T_1 or are generated during $[T_1, T_2]$. By $Assum_3$, we have $n_r \leq k+\ g \cdot (T_2 - T_1)$.
- $timer_0^{phase}$ goes off because the timer ticked down to 0. Since $timer_0^{phase}$ is never

set to any values other than TM_{phase} and 0, and it is only set to 0 in a $receive_0^m$ event, at least TM_{phase} time elapses between any two consecutive times that the timer goes off due to ticking down, so it goes off at most $\lceil \frac{T_2-T_1}{TM_{phase}} \rceil$ times in $[T_1, T_2]$.

So in $[T_1, T_2]$ the number of $start\_phase_0$ events is no more than $k+$ $g \cdot (T_2 - T_1)$ $+ \lceil \frac{T_2-T_1}{TM_{phase}} \rceil$. ■

Lemma 5 *Starting from an arbitrary configuration* c_0*, within time* $2M_{lt}$ *there is a configuration satisfying* $conv_{token}$.

Proof. By Lemma 1, there is a configuration c_1 in which all the m-$token$s are generated by p_0 after c_0, and a later configuration c_2 in which all the m-$token$s are generated by p_0 after c_1. We will prove c_2 satisfies $conv_{token}$. Suppose c_2 is in phase k. By Lemma 1, c_2 occurs within $2M_{lt}$ from c_0.

By Lemma 1, all the m-$token$s in a configuration c after c_1 are generated within time M_{lt} before c, thus the number of m-$token$s in c is no more than $g \cdot M_{lt}$ by $Assum_3$. So (1) of definition 5 is proved for c_2. Given any m-$token$ t in c_2, assume t first occurs in configuration c_t in phase k_t. Since c_t is after c_1, the number of m-$token$s in c_t is no more than $g \cdot M_{lt}$. Let T be the time interval between c_2 and c_t. We have $T \leq d \cdot max(t.ts) \leq M_{lt}$ by Lemma 1, and $k - k_t \leq g \cdot M_{lt} + g \cdot T + \lceil \frac{T}{TM_{phase}} \rceil$ $\leq g \cdot M_{lt} + g \cdot d \cdot max(t.ts) + 1 \leq 2 \cdot g \cdot M_{lt} + 1 < M_{tid}$ by Lemmas 1 and 4. Thus m-$token$s with the same id are generated in the same phase by (3) of Lemma 3 and (2), (3) and (4) of definition 5 follow. ■

Lemma 6 *Given any execution starting from a configuration* c_0 *satisfying* $conv_{token}$*, every configuration* c *in* σ *satisfies* $token\_safe$.

Proof. We denote the time interval between c and c_0 by T and consider any m-$token$ t in c. Assume c is in phase k.

If t is generated between c and c_0, say at time T_t in phase k_t, we have $T - T_t \leq M_{lt}$ by Lemma 1. Since all the tokens at T_t either exist in c_0 or are generated within M_{lt} before T_t by Lemma 1, the number of tokens at T_t is no more than $2g \cdot M_{lt}$ by $Assum_3$. By Lemma 4, we have $k - k_t \leq 2g \cdot M_{lt} + g \cdot M_{lt} + 1 \leq M_{tidr}$. Thus in this case we have $t.id = tid_0^{k_t} \in [tid_0^k - M_{tidr}, tid_0^k]$ by Lemma 3. Otherwise we denote $max(t.ts)$ in c_0 by $max_{ts}^{c_0}$ and in c by max_{ts}^{c}, and $g \cdot M_{lt} + g \cdot d \cdot max_{ts}^{c_0} + 1$

by x. We have $T \leq d \cdot (L - max_{ts}^{c_0}) \leq M_{lt}$ by Lemma 1 and $k \leq g \cdot M_{lt} + g \cdot T$ $+ \lceil \frac{T}{TM_{phase}} \rceil \leq g \cdot M_{lt} + g \cdot d \cdot (L - max_{ts}^{c_0}) + 1 \leq M_{tidr}$ by Lemma 4. By (2) of $conv_{token}$, we have $t.id \in [tid_0^0 - x, tid_0^0]$, which is a subset of $[tid_0^0 - x, tid_0^0 + k]$ $= [tid_0^k - k - x, tid_0^k] \subseteq [tid_0^k - M_{tidr}, tid_0^k]$ because $k + x \leq M_{tidr}$. Thus (1) is true in both cases.

Now we prove (2). First we consider the case that every $m\text{-}token$ in c is generated after c_0. Assume $m\text{-}token$ t is generated in phase k_t. We know $k_t \in [k - M_{tidr}, k]$ from above. Because $M_{tidr} \leq M_{tid}$, $m\text{-}token$s generated in different phases have different token ids by Lemma 3, thus (2) is true.

Otherwise we know $k = tid_0^k - tid_0 \leq M_{tidr}$ from above. So in c, all the $m\text{-}token$s existing in c_0 have id in $R_1 = [tid_0^0 - M_{tidr}, tid_0^0]$. By (2) of definition 5 all the $m\text{-}token$s generated after c_0 have ids in $R_2 = [tid_0^0, tid_0^k] \subseteq [tid_0^0, tid_0^0 + M_{tidr}]$. Consider all the $m\text{-}token$s with id $l \in R_1 \cup R_2$ in c. If $l \in [tid_0^0 - M_{tidr}, tid_0^0 - 1]$, all these $m\text{-}token$s exist in c_0 and they have the same $ring$ and new_set by (3) of definition 5. If $l = tid_0^0$, these $m\text{-}token$s either exist in c_0 or are generated in phase 0 with $ring = ring_0^0$ and $new\_set = new\_set^0$, so (2) is true in this case by (4) of definition 5; otherwise they are generated in the same phase after c_0 and (2) is still true. ■

An execution in which all the configurations satisfy $token\_safe$ exhibits good behaviors. We state the conditions under which a processor's token id remains unchanged in Lemma 7.

Lemma 7 *Let σ be an execution in which all the configurations are $token\_safe$. For each processor p_i and all $k \geq 0$, we have:*

(1) *If $(tid_i = tid_0)$ or $(tid_i = tid_0 - 1)$ in configuration c, then line 9.10 is $false$ in the $receive_i^m(t)$ event following c.*

(2) *If in phase k there exists a configuration c in which*
   (2.1) $(status_i = joined) \wedge (tid_i = tid_0^k)$*, or*
   (2.2) $(status_i = joined) \wedge (\ tid_i \in \{\ M_{tid}, 0, \ldots, M_{tidr}\ \}) \wedge$
      $(tid_i \notin [tid_0^k - M_{tidr} - 1, tid_0^k - 1])$
   then tid_i remains unchanged after c in phase k.

(3) *If in phase k there exists a configuration c before c_g^k in which $(status_i = joined)$ $\wedge\ (tid_i = tid_0^k - 1)$, then tid_i remains unchanged between c and c_g^k (including c_g^k).*

Proof. Proof of (1): If $t.id \neq 0$, line 9.10 is $false$. Otherwise since $t.id = 0 \in [tid_0 - M_{tidr}, tid_0]$ by definition 6, we have $tid_0 \in [0, M_{tidr}]$, that is, $tid_i \in \{M_{tid}, 0, \ldots, M_{tidr}\}$, so line 9.10 is $false$.

Proof of (2): If (2.1) is true in c, then line 9.2 is $false$, line 9.10 is $false$ by part (1), which was just proved, and line 9.13 is $false$ because no $m\text{-}token$ with id $tid_0^k + 1$ exists by definition 6. If (2.2) is true in c, then line 9.2 and line 9.10 are $false$, and line 9.13 is $false$ because all the $m\text{-}token$s have id in $[tid_0^k - M_{tidr}, tid_0^k]$ by definition 6. So (2) is proved.

Proof of (3): If $status_i = joined$, then line 9.2 is $false$. If $tid_i = tid_0^k - 1$, then line 9.10 is $false$ by part (1), which has already been proved. Since no $m\text{-}token$ with id tid_0^k exists before e_g^k in phase k by definition 6, line 9.13 is $false$ between c and c_g^k. So (3) is proved. ■

Now we consider the generation of current tokens of phase $k \geq 1$ in such an execution.

Lemma 8 *Consider any execution σ in which all the configurations are $token\_safe$. For each phase $k \geq 1$ of σ, we have:*

(1) *All the current tokens of phase k are generated in phase k with the same $ring$ and new_set.*

(2) *The time interval between e_g^k and e_s^k is no more than M_{lt}, and there is no $m\text{-}token$ in the configuration previous to e_g^k if $(discon_0 = true)$ in e_s^k.*

Proof. Since $k \geq 1$, the c_s^k configuration exists, which is $token\_safe$. So there is no $m\text{-}token$ with id tid_0^k in c_s^k by definition 6 and (1) is proved.

Now we prove (2). First we prove at least one $generate_0^m$ event occurs in phase k. Otherwise no $m\text{-}token$ with id tid_0^k exists in phase k by part (1) (which was just proved), so $timer_0^{phase}$ is never reset in a $receive_0^m$ event and e_s^{k+1} is activated only when $timer_0^{phase}$ goes off. Thus the length of phase k is TM_{phase}. Since $timer_0^{gen}$ is set to 0 or to M_{lt}(which is less than TM_{phase}) in e_s^k, it goes off and a $generate_0^m$ event is activated in phase k, which is a contradiction.

Since $timer_0^{gen}$ is not reset between e_g^k and e_s^k, the time interval between e_g^k and e_s^k is equal to its reset value in e_s^k, which is M_{lt} if $(discon_0 = true)$ and 0 otherwise. If $(discon_0 = true)$, no $m\text{-}token$ exists in the configuration previous to e_g^k by

Lemma 1. ■

Now we consider membership maintenance in such an execution.

Lemma 9 *Consider an execution σ and phase $k \geq 1$ of σ, such that all the configurations in σ are $token\_safe$. The following are true for each $p_i \in ring_0^{k+1}$.*

(1) *There is a $receive_i^m(t)$ event, denoted by e_i^v, in phase k, such that $t.id = tid_0^k$ and p_i is added to $t.visited\_set$ in e_i^v. Furthermore, $status_i = joined$ in any configuration after e_i^v in phase k and phase $k+1$.*
(2) *If $p_i \notin ring_0^k$, then $tid_i = tid_0^k$ in c_s^{k+1}.*

Proof. By line 4.3, we have $p_i \in visited\_set_0$ in c_s^{k+1}. Notice $visited\_set_0$ is set to $\{0\}$ in e_s^k, and it is updated in phase k according to $visited\_set$ in the current tokens received by p_0, which are generated in phase k by Lemma 8. So there exists a current token which includes p_i in its $visited\_set$. Since every $m\text{-}token$'s $visited\_set$ is set to $\{0\}$ when generated, there exists such an e_i^v event in phase k. Right after e_i^v, $status_i$ is $joined$. The only code changing $status_i$ to $joining$ is a $send_i^j$ event. Since $timer_i^{join}$ is set to TM_{join} in e_i^v, after which $timer_i^{join}$ keeps going down unless it is reset to TM_{join} in a $receive_i^m$ event, it takes at least TM_{join} for $timer_i^{join}$ to reach 0. Thus a $send_i^j$ event will not occur within TM_{join} from e_i^v. Because $TM_{join} > 2TM_{phase}$, a $send_i^j$ event will not occur after e_i^v in phase k and phase $k+1$ by Lemma 3. So (1) is proved.

If $p_i \notin ring_0^k$, p_i can be added to $t.visited\_set$ only in line 9.5, so we have $tid_i = t.id = tid_i^k$ and $status_i = joined$ in the configuration following e_i^v. After that, tid_i remains unchanged in phase k by (2.1) of Lemma 7. So (2) is proved. ■

8.3 Safe Properties of Non-member

In this section, we discuss the properties of non-members. First we define the appropriate levels of information related to a non-member processor according to how long ago the processor left the system (definition 7). Lemma 10 shows the time required to move from one level to the next. We prove in Lemmas 11 and 12 that eventually the information related to a non-member will be kept in an appropriate level and starting from that point, property $nmem\_safe$ (definition 8) is always guaranteed, that is, a processor rejoin is allowed to the system only when there is no old information relating to it still in the system. Thus we guarantee that when a processor rejoins the system, it cannot enter the critical section before it is initialized by a current token (Lemma 13).

First we give definitions of $conv_{nmem}$ and $nmem\_safe$.

Definition 7 *($conv_{nmem}$). Given a configuration c, c satisfies property $conv_{nmem}$ if and only if for every processor p_i with $(member_i = false)$ in c, we have:*

(1) *if* $(timer_i^{app} \leq 2M_{lt} + 2TM_{phase})$ *then* $(pred_1 = true)$
(2) *if* $(timer_i^{app} \leq M_{lt} + 2TM_{phase})$ *then* $(pred_2 = true)$
(3) *if* $(timer_i^{app} \leq M_{lt} + TM_{phase})$ *then* $(pred_3 = true)$
(4) *if* $(timer_i^{app} \leq M_{lt})$ *then* $(pred_4 = true)$

where

- $pred_1 \equiv (\ \nexists\ m\text{-}token\ t,\ p_i \in t.visited\_set\ ) \wedge (\nexists\ j\text{-}token\ t,\ t.pid = p_i)$
- $pred_2 \equiv (p_i \notin visited\_set_0) \wedge (p_i \notin join\_set_0)$
- $pred_3 \equiv (p_i \notin ring_0) \wedge (p_i \notin new\_set_0)$
- $pred_4 \equiv \forall\ m\text{-}token\ t,\ (p_i \notin t.ring) \wedge (p_i \notin t.new\_set)$

Definition 8 *($nmem\_safe$). A configuration is $nmem\_safe$ if and only if for every p_i with $(member_i = false) \wedge (timer_i^{app} = 0)$, $(pred_3 \wedge pred_4)$ is $true$ for p_i.*

The level of information related to p_i represented by each predicate is decreasing. The following lemmas shows the time required to move from one level to the next.

Lemma 10 *Given $i > 0$ and any execution σ in which $(member_i = false)$ at time T, we have the following properties, where $T_a = T + M_{lt}$, $T_b = T_a + TM_{phase}$, $T_c = T_b + TM_{phase}$, and $T_d = T_c + M_{lt}$.*

(0) $\forall i$, $1 \leq i \leq 4$, *if* $\wedge_{j=1}^{i} pred_j$ *is true at* T, *then* $\wedge_{j=1}^{i} pred_j$ *remains* $true$ *as long as* $member_i$ *remains* $false$.
(1) *If* $member_i$ *is* $false$ *in* $[T, T_a]$, *then* $pred_1$ *is true at* T_a.
(2) *If* $pred_1$ *is* $true$ *at* T_a *and* $member_i$ *is* $false$ *in* $[T_a, T_b]$, *then* $pred_2$ *is true at* T_b.
(3) *If* $(pred_1 \wedge pred_2)$ *is* $true$ *at* T_b *and* $member_i$ *is* $false$ *in* $[T_b, T_c]$, *then* $pred_3$ *is* $true$ *at* T_c.
(4) *If* $(pred_1 \wedge pred_2 \wedge pred_3)$ *is* $true$ *at* T_c *and* $member_i$ *is* $false$ *in* $[T_c, T_d]$, *then* $pred_4$ *is* $true$ *at* T_d.

Proof. Suppose T_b is in phase k_b and T_c is in phase k_c. We notice that (a) p_i does not send any $j\text{-}token$ and p_i is not added to any $m\text{-}token$'s $visited\_set$ as long as $member_i$ remains $false$; (b) p_i can be added to $join\_set_0$ ($visited\_set_0$ resp.) only if there exists $j\text{-}token$ from p_i ($m\text{-}token\ t$ with $p_i \in t.visited\_set$ resp.); (c) p_i can be added to $ring_0$ (new_set_0 resp.) only if p_i has been added to $visited\_set_0$ ($join\_set_0$ resp.) ; (d) p_i is included in a $ring$ (new_set resp.) of a $m\text{-}token\ t$ only if $p_i \in ring_0$ ($p_i \in new\_set_0$ resp.) when t is generated.

If $pred_1$ is $true$ at T, it is $true$ as long as $member_i$ is $false$ by (a); if $(pred_1 \wedge pred_2)$ is $true$ at T, it is $true$ as long as $member_i$ is $false$ by (a) and (b); if $(\ pred_1 \wedge pred_2 \wedge pred_3\ )$ is $true$ at T, it is $true$ as long as $member_i$ is $false$ by (a), (b) and (c); if $(\ pred_1 \wedge pred_2 \wedge pred_3 \wedge pred_4\ )$ is $true$ at T, it is $true$ as long as $member_i$ is $false$ by (a), (b), (c) and (d). So (0) is proved.

Since all tokens existing at T have been discarded by T_a by Lemma 1, (1) can be proved by (a). Since event $e_s^{k_b}$ exists in $[T_a, T_b]$ by Lemma 3, in which $visited\_set_0$ and $join\_set_0$ are reset, (2) can be proved by (b) because $pred_1$ is $true$ in $[T_a, T_b]$ by case (0). Since by Lemma 3 configuration $c_s^{k_c}$ exists in $[T_b, T_c]$, (3) can be proved by (c) because $pred_2$ is $true$ in $c_s^{k_c}$ by case (0). Since all the tokens existing at T_c have been discarded by T_d by Lemma 1, (4) can be proved by (d) because $pred_3$ is $true$ in $[T_c, T_d]$ by cases (0). ∎

Lemma 11 *Starting from an arbitrary configuration, within time* $2M_{lt}+3TM_{phase}$*, there is a configuration satisfying* $conv_{nmem}$.

Proof. We will prove the configuration c at time $T = 2M_{lt} + 3TM_{phase}$ from the initial configuration satisfies $conv_{nmem}$. Consider any processor p_i with $member_i = false$ in c. If there is no $leave_i$ event before c, then $member_i$ has remained $false$ for T time, so c satisfies $conv_{nmem}$ by (1), (2), (3), (4) and (0) of Lemma 10.

Otherwise let e be the last $leave_i$ event before c. In e, $timer_i^{app}$ is set to TM_{app}. Denote the value of $timer_i^{app}$ in c by T', so in c, $member_i$ has been $false$ for $T_f = TM_{app} - T'$ time. In c, if $T' \leq 2M_{lt} + 2TM_{phase}$, that is, $T_f \geq TM_{phase} > M_{lt}$, $pred_1$ is $true$ by (1) and (0) of Lemma 10; if $T' \leq M_{lt} + 2TM_{phase}$, that is, $T_f \geq M_{lt} + TM_{phase}$, $pred_2$ is $true$ by (1), (2) and (0) of Lemma 10; if $T' \leq M_{lt} + TM_{phase}$, that is, $T_f \geq M_{lt} + 2TM_{phase}$, $pred_3$ is $true$ by (1), (2), (3) and (0) of Lemma 10; if $T' \leq M_{lt}$, that is, $T_f \geq 2M_{lt} + 2TM_{phase}$, $pred_4$ is $true$ by (1), (2), (3), (4) and (0) of Lemma 10. ∎

Lemma 12 *Given any execution σ starting from a configuration c_0 satisfying $conv_{nmem}$, every configuration c in σ is $nmem\_safe$.*

Proof. Consider any processor p_i such that $member_i = false$ and $timer_i^{app} = 0$ in c. If there is a $leave_i$ before c in σ, then $member_i$ has been $false$ for at least $TM_{app} \geq 2TM_{phase} + 2M_{lt}$ time by c, so the claim is true by (1), (2), (3), (4) and (0) of Lemma 10. Otherwise, $member_i$ is $false$ between c and c_0. Let T be the value of $timer_i^{app}$ in c_0. Since $timer_i^{app} = 0$ in c, $member_i$ has been $false$ for T time by c. If $T > 2TM_{phase} + 2M_{lt}$, the claim is proved by (1), (2), (3), (4) and (0) of Lemma 10. If $T \in [M_{lt} + 2TM_{phase}\ ,\ 2M_{lt} + 2TM_{phase}\ ]$, then $pred_1$ is $true$ in c_0 by definition 7 and the claim can be proved by (2), (3), (4) and (0) of Lemma 10. If $T \in [M_{lt} + TM_{phase}, M_{lt} + 2TM_{phase}]$, then $(pred_1 \wedge pred_2\ )$ is $true$ in c_0 by definition 7 and the claim can be proved by (3), (4) and (0) of Lemma 10. If $T \in [M_{lt}, M_{lt} + TM_{phase}]$, then $(pred_1 \wedge pred_2 \wedge pred_3\ )$ is $true$ in c_0 by definition 7 and the claim can be proved by (4) and (0) of Lemma 10. If $T \in [0, M_{lt}]$, then $(pred_3 \wedge pred_4)$ is $true$ in c_0 by definition 7 and the claim can be proved by (0) of Lemma 10. ∎

From the following lemma, we can see that eventually a processor not in $ring_0$ will not enter the critical section.

Lemma 13 *Given an arbitrary configuration c_0, consider any execution σ starting from c_0. Within time $T = 3M_{lt} + 5TM_{phase}$, there exists an $e_s^{k_c}$ event, $k_c > 1$, such that*

(1) all configurations after $e_s^{k_c}$ are $token\_safe$ and $nmem\_safe$, and

(2) given any $k \geq k_c$, in any configuration between e_g^k and e_g^{k+1}, for any $p_i \notin ring_0^k$, we have

(2.1) $(member_i = false)$ or $(\forall t, p_i \notin t.ring)$, and

(2.2) p_i does not enter the critical section.

Proof. By Lemmas 5, 6, after time $2M_{lt}$, all the configurations are $token\_safe$. By Lemmas 11 and 12, after time $2M_{lt} + 3TM_{phase}$, all the configurations are $nmem\_safe$. Thus after time $T' = 2M_{lt} + 3TM_{phase}$, all the configurations are $token\_safe$ and $nmem\_safe$. By Lemma 3, there exists some $k_0 \geq 1$ such that the $e_s^{k_0}$ event occurs in $[T', T'']$ where $T'' = T' + TM_{phase}$, and k_c such that the $e_s^{k_c}$ event occurs in $[T'' + M_{lt}, T]$. Thus (1) is true for k_0 and k_c.

Now we prove (2.1) is true and we can get (2.2) directly from (2.1). Let k_1 be the smallest number such that $p_i \notin ring_0^l$ for any $l \in [k_1, k]$ (informally, phase k_1 to phase k is a maximal execution segment in which $p_i \notin ring_0$.) Let σ_1 be the execution of phase $k_1 - 1$ if $k_1 > k_0$, otherwise σ_1 is empty. Let σ_2 be the execution from $e_s^{max\{k_1,k_0\}}$ to e_g^{k+1}. Thus all the configurations in σ_1 or σ_2 are $token\_safe$ and $nmem\_safe$. We notice that (a) if σ_1 is not empty, that is $k_1 > k_0$, we have $p_i \in ring_0^{k_1-1}$ in σ_1 and there is no $join_i$ event in σ_1 by definition 8; (b) for any $m\text{-}token$ t generated in σ_2 we have $p_i \notin t.ring$; and (c) all the $m\text{-}token$s in phase k are generated after $e_s^{k_0}$ by Lemma 1. We have the following two cases.

- If there is a $join_i$ event e_{join} in σ_2, let c_{join} be the configuration previous to e_{join}, in which $(member_i = false) \wedge (timer_i^{app} = 0)$. By definition 8, there is no $m\text{-}token$ including p_i in $ring$ in c_{join}. So there is no $m\text{-}token$ including p_i in $ring$ in all the configurations after e_{join} in σ_2 by (b). If e_{join} occurs before e_g^k, the claim is proved. Otherwise e_{join} occurs after e_g^k. We notice the time interval between two $join_i$ and $leave_i$ events is at least TM_{app} because $timer_i^{app}$ is set to TM_{app} in a $leave_i$ event and a $join_i$ event occurs only if $timer_i^{app} = 0$. So there is no $leave_i$ event between e_g^k and e_{join} by Lemma 3 and 8. Thus $member_i$ remains $false$ between e_g^k and e_{join} and the claim is proved in this case.
- Otherwise by (a) no $join_i$ event occurs in σ_1 and σ_2. If $member_i = false$ in the configuration previous to e_g^k, then $member_i$ remains $false$ in σ_2 and the claim is proved. Otherwise $member_i$ is always $true$ in σ_1 and σ_2. We consider the following two cases:
  - If σ_1 is empty, that is, $k_1 \leq k_0$, then σ_2 is the execution between $e_s^{k_0}$ and e_g^{k+1}. Thus all the tokens in phase k are generated in σ_2 by (c) and in phase k there is no token which includes p_i in $ring$ by (b). Thus (2.1) is proved in this case.
  - Otherwise we have $k_1 \geq k_0 + 1$, so all the configurations in phase $k_1 - 1$ are $token\_safe$. Since $member_i$ is $true$ in phase $k_1 - 1$, we have $p_i \notin t.leave\_set$ for any current token t of phase $k_1 - 1$, which is generated in this phase by Lemma 8. Because $k_1 \geq k_0 + 1 \geq 2$, the $e_s^{k_1-1}$ event exists, in which $leave\_set_0$ is reset. Since $leave\_set_0$ is updated according to the $leave\_set$ on the current tokens received by p_0 in phase $k_1 - 1$, we have $p_i \notin leave\_set_0$ in $e_s^{k_1}$. Because $p_i \notin ring_o^{k_1}$, we have $p_i \notin visit\_set_0$ in $e_s^{k_1}$. So in $e_s^{k_1}$, $discon_0$ is $true$ and no $m\text{-}token$ exists in the configuration previous to $e_g^{k_1}$ by Lemma 8. So after e_g^{k-1}, all the $m\text{-}token$s are generated after $e_g^{k_1}$ and no $m\text{-}token$ includes p_i in $ring$ by (b). Thus the claim is proved.

■

8.4 Eventual Convergence

In this section, we define a property $conv_{member}$ of configurations. Based on Lemma 14, we show in Lemma 15 that, starting from an arbitrary configuration, there exists a phase k_c, such that $c_s^{k_c}$ satisfies $conv_{member}$ and all the configurations after $c_s^{k_c}$ are $token\_safe$ and $nmem\_safe$.

The intuition for $conv_{member}$ is that, in a configuration satisfying $conv_{member}$, all the members indicate they have joined the system by setting $status_i$ to $joined$, they have a correct token id, which is equal to tid_0, and they have been visited recently, that is, $timer_i^{join} \geq TM_{phase} + M_{lt}$.

Definition 9 *($conv_{member}$). A configuration is $conv_{member}$ if and only if for any $p_i \in ring_0$, we have $(status_i = joined) \wedge (tid_i = tid_0) \wedge (timer_i^{join} \geq TM_{phase} + M_{lt})$.*

In Lemma 14, we show that, starting from a phase with $tid_0 = 0$, the range of the members' token ids is decreased by one in each phase.

Lemma 14 *Consider an execution σ, in which all the configurations are $token\_safe$, and a phase b of σ, such that $tid_0^b = 0$ and $b \geq 1$. For any $p_i \in ring_i^{b+k+1}$, $0 \leq k \leq M_{tidr}$, we have $tid_i \in [k, M_{tidr}]$ in c_s^{b+k+1}.*

Proof. We have $tid_0^{b+k} = k$ and $tid_0^{b+k+1} = k + 1$ by Lemma 3. Given $p_i \in ring_i^{b+k+1}$, by Lemma 9 the e_i^v event exists in phase $b + k$ and we have $status_i = joined$ after e_i^v in phase $b + k$ and phase $b + k + 1$. Note that if we have $tid_i \in [k, M_{tidr}]$ in the configuration c_i^v following e_i^v, then tid_i remains unchanged in phase $b+k$ by (2.2) of Lemma 7. So in the rest we only need to prove $tid_i \in [k, M_{tidr}]$ in c_i^v. We prove it by induction on k.

In phase b, if p_i is added to $t.visited\_set$ by line 9.5 in e_i^v, we have $tid_i = t.id = 0 \in [0, M_{tidr}]$ in c_i^v. Otherwise we have $tid_i \in \{M_{tid}, 0, \ldots, M_{tidr}\}$ at line 9.13 because $t.id = 0$, after which tid_i is updated to 0 if $tid_i = M_{tid}$ by lines 9.13-9.15. So we have $tid_i \in [0, M_{tidr}]$ in c_i^v. Thus the claim is true for $k = 0$.

Assume the claim is true for l, $0 \leq l \leq k - 1$. We show it holds for k. If $p_i \notin$

$ring_0^{b+k}$, we have $tid_i = tid_0^{b+k} = k \in [k, M_{tidr}]$ in c_s^{b+k+1} by Lemma 9. Otherwise we have $p_i \in [k-1, M_{tidr}]$ in c_s^{b+k} by the inductive hypothesis, and $status_i = joined$ in phase $b+k$ by Lemma 9. Note if $tid_i \in [k-1, M_{tidr}]$ is true in the configuration previous to a $receive_i^m(t')$ event in phase $b+k$, then it is true at the end of this event for the following reason. Since line 9.2 and 9.10 are $false$, tid_i can be changed only when line 9.13 is true. Since we have $tid_i \in [k-1, M_{tidr}]$, $t'.id \in [k - M_{tidr}, k]$ by definition 6, and $[k, M_{tidr}+1] \cap [k - M_{tidr}, k] = \{k\}$, line 9.13 is true only if $t'.id = tid_i + 1 = k$, and tid_i is updated to $k \in [k-1, M_{tidr}]$ if it is true.

Since we have $tid_i \in [k-1, M_{tidr}]$ in c_s^{b+k}, $tid_i \in [k-1, M_{tidr}]$ is true in the configuration previous to all $receive_i^m$ events in phase $(b+k)$, including e_i^v. Since we have $t.id = k$ in e_i^v, tid_i is updated to k in line 9.14 if $tid_i = k-1$. So we have $tid_i \in [k, M_{tidr}]$ in c_i^v, and the claim is true for k. ■

Theorem 15 *Given an arbitrary configuration* c_0*, consider any execution* σ *starting from* c_0*. Within time* $2M_{lt} + 3TM_{phase} + (M_{tid} + M_{tidr}) \cdot TM_{phase}$*, there exists an* $e_s^{k_c}$ *event, for some* $k_c > 1$*, such that*

(1) all configurations after c_s^t *are* $token\_safe$ *and* $nmem\_safe$ *and*
(2) $c_s^{k_c}$ *satisfies* $conv_{member}$*.*

Proof. By Lemmas 5, 6, after time $2M_{lt}$, all the configurations are $token\_safe$. By Lemmas 11 and 12, after time $2M_{lt} + 3TM_{phase}$, all the configurations are $nmem\_safe$. Thus after time $T = 2M_{lt} + 3TM_{phase}$, all the configurations are $token\_safe$ and $nmem\_safe$.

By Lemma 3, after T there exists a phase b such that $tid_0^b = 0$ and $b \geq 1$. Let m be $b + M_{tidr}$. We will prove $c_s^{k_c}$ satisfies $conv_{member}$, where $k_c = m+1$. By Lemma 3, 5 and 11, we can see the time of $e_s^{k_c}$ is no more than $2M_{lt} + 3TM_{phase} + (M_{tid} + M_{tidr}) \cdot TM_{phase}$ after the initial configuration.

For any $p_i \in ring_0^{m+1}$, by Lemma 9 the e_i^v event exists in phase m and we have $status_i = joined$ after e_i^v in phase m and phase $m+1$, including c_s^{m+1}. By Lemma 14 we have $tid_i = M_{tidr} = tid_0^m$ in c_s^{m+1}. Since $timer_i^{join}$ is set TM_{join} in e_i^v, we have $timer_i^{join} \geq TM_{join} - TM_{phase} \geq TM_{phase} + M_{lt}$ in c_s^{m+1} by Lemma 3 and 8. So c_s^{m+1} satisfies $conv_{member}$. ■

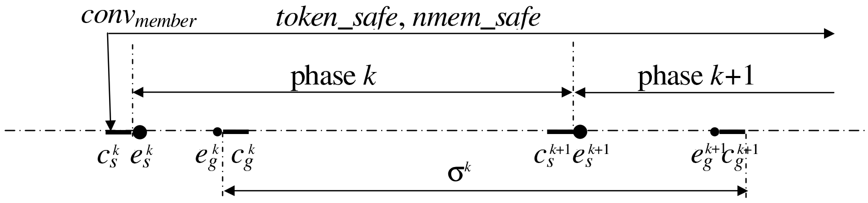

Fig. 8. Proof of $Mutual\ Exclusion$ after convergence

In the sequel, we say an execution has *converged* if it has reached a configuration $c_g^{k_c}$, where k_c satisfies the properties in Lemma 13 and Theorem 15.

8.5 Mutual Exclusion after Convergence

In this section we focus on an execution σ which has converged. We denote by σ^k the subsequence between c_g^k and c_g^{k+1} of σ, including c_g^k and c_g^{k+1} (see Fig. 8). By Lemma 13, we know that only processors in $ring_0^k$ can enter the critical section in σ^k. In this section, we show that if c_s^k satisfies $conv_{member}$, then no more than one processor is in the critical section at the same time in σ^k (Lemma 16), and c_s^{k+1} satisfies $conv_{member}$ (Lemma 17). Thus $Mutual\ Exclusion$ can be guaranteed from c_g^k (Theorem 18).

Lemma 16 *Given an execution σ and $k > 1$, such that c_s^k satisfies $conv_{member}$, and all the configurations after c_s^k are $token\_safe$ and $nmem\_safe$, we have in σ^k:*

(1) *For any $p_i \in ring_0^k$,*

(1.1) *p_i is in the critical section only when being visited by an m-$token$ with id tid_0^k the first time in σ^k.*

(1.2) *if p_i never enters the critical section in σ^k, tid_i remains unchanged. Otherwise let c_{cs} be the last configuration in σ^k in which p_i is in the critical section. We have $tid_i = tid_0^k - 1$ in all configurations in σ^k previous to c_{cs} and including c_{cs}, and $tid_i = tid_0^k$ in all configurations in σ^k after c_{cs}.*

(2) *No more than one processor in $ring_0^k$ is in the critical section at the same time in any configuration in σ^k.*

Proof. By definition 9, we have $status_i = joined$ and $tid_i = tid_0 = tid_0^k - 1$ in c_s^k. First we notice $status_i$ remains $joined$ in σ^k for the following reason: $status_i$ is set to $joining$ only in a $send_i^j$ event, which will not occur in σ^k because $timer_i^{join}$ will not reach 0 in σ^k by Definition 9 and Lemmas 3 and 8. We denote the first event that updates tid_i in σ^k by e_i^u. By (3) of Lemma 7, tid_i remains unchanged before c_g^k. So we have $tid_i = tid_0^k - 1$ in c_g^k and line 9.10 is $false$ in e_i^u by (1) of Lemma 7. Since line 9.2 is always $false$ because $status_i = joined$, line 9.13 is $true$ in e_i^u, that is, p_i is visited by an $m\text{-}token$ with id tid_0^k and tid_i is updated to tid_0^k by line 9.14 in e_i^u. Since line 9.13 is always $true$ when p_i is visited by an $m\text{-}token$ with id tid_0^k if $(tid_i = tid_0^k - 1)$, this is the first time p_i is visited by an m-$token$ with id tid_0^k. Once tid_i is updated to tid_0^k, tid_i remains unchanged in phase k by (2.1) of Lemma 7 and in phase $(k+1)$ before e_g^k by (3) of Lemma 7. Thus (1) is proved.

Now we prove (2). Assume in contradiction that p_i and p_j are in the critical section at the same time, say in configuration c. Let $m\text{-}token$ t_i (t_j resp.) be the first token in $queue_i$ ($queue_j$ resp.) in c. By part (1.1), which was just proved, t_i (t_j resp.) is the first token with id tid_0^k which visits p_i (p_j resp.). Since any $m\text{-}token$ with id tid_0^k is generated in phase k by Lemma 8, we have $t_i.ring = t_j.ring = ring_0^k$. By the predicate for the critical section, we also have in c: (a) p_i is the first processor in $t_i.ring$ not in $t_i.visited\_set$, and (b) p_j is the first processor in $t_j.ring$ not in $t_j.visited\_set$. Without loss of generality, we assume p_i is before p_j in $ring_0^k$, thus we have $p_i \in t_j.visited\_set$ by (b). So p_i is visited by t_j before p_i is visited by t_i. Contradiction! ∎

Lemma 17 *Given an execution σ, if there exists a phase $k > 1$, such that c_s^k satisfies $conv_{member}$, and all the configurations after c_s^k are $token\_safe$ and $nonmember_{safe}$, then c_s^{k+1} satisfies $conv_{member}$.*

Proof. Given any $p_i \in ring_0^{k+1}$, by Lemma 9 we have an event e_i^v in phase k such that $status_i = joined$ after e_i^v in phase k and phase $k+1$, including c_s^{k+1}. Since $timer_i^{join}$ is set to $TM_{join} \geq 2TM_{phase} + 2M_{lt}$ in e_i^v, we have $timer_i^{join} \geq TM_{phase} + M_{lt}$ in c_s^{k+1} by Lemmas 3 and 8. Now we consider tid_i in c_s^{k+1}. If $p_i \notin ring_0^k$, we have $tid_i = tid_0^k$ in c_s^{k+1} by Lemma 9. Otherwise we have $p_i \in ring_0^k$. Since p_i is visited by a current token in e_i^v in phase k, we have $tid_i = tid_0^k$ in c_s^k by (1.2) of Lemma 16. So c_s^{k+1} satisfies $conv_{member}$. ∎

By Lemmas 13, Lemma 16 and Lemma 17, we have:

Theorem 18 *$Mutual\ Exclusion$ is satisfied when an execution has converged.*

Since one of the conditions for processor p_i to enter the critical section is being visited by a token, the level of progress provided by this algorithm depends on the frequency with which a processor is visited by tokens, that is, LRV_L's performance for token circulation. The performance of LRV_L is affected by many factors, including message loss, choice of L, and the mobility pattern of the network. In this section, we discuss the different levels of requirements on LRV_L's performance to achieve different levels of progress. We define the following levels of LRV_L's performance, in which R_0 is required to guarantee that p_0 can receive p_i's $join\_request$ eventually, $R_1(i)$ is used to guarantee that p_i enters the critical section infinitely often, and R_2 is used to guarantee $Bounded\ Waiting$.

- $R_0(i)$: If infinitely many tokens are generated by p_i, then infinitely many of them arrive at p_0 within L hops.
- $R_1(i)$: If infinitely many tokens that include p_i in $ring$ or new_set are generated by p_0, then infinitely many of them visit p_i and come back to p_0 within L hops.
- R_2: At least one of every $(TM_{phase} - M_{lt}) \cdot g$ tokens generated by p_0 visits all the processors in their new_set and $ring$ within L hops.

Lemma 19 *Consider phase k of any execution which has converged. Let T^k be the time of event e_s^k. Let processor p_i and m-token t_m be such that t_m is generated in $[T^k, T^k + TM_{phase} - M_{lt}]$, p_i is in $t_m.new\_set$ or $t_m.ring$ and t_m visits p_i and comes back to p_0 within L hops. Then we have $p_i \in ring_0^{k+1}$ and p_i enters the critical section in phase k if $p_i \in t_m.ring$.*

Proof. By Lemma 8 any current token t of phase k, including t_m, satisfies $(t.ring = ring_0^k) \wedge (t.new\_set = new\_set_0^k)$. By Lemma 1, token t_m comes back to p_0 after visiting p_i within $[T^k, T^k + TM_{phase}]$. Since phase k is ended only if a current token comes back to p_0 after visiting all the processors in $new\_set_0 \cup ring_0$, including p_i, or phase k has lasted for time TM_{phase}, p_i is visited by a current token and added to $visited\_set_0$ in phase k. So we have $p_i \in ring_0^{k+1}$ and p_i enters the critical section if $p_i \in t.ring$ by (1.1) of Lemma 16. ■

Theorem 20 *Consider any execution that has converged.* (1) *If $R_0(i)$ and $R_1(i)$ are satisfied for some processor p_i, then $No\ Deadlock$ is guaranteed;* (2) *if $R_0(i)$ and $R_1(i)$ are satisfied for every processor p_i, then $No\ Lockout$ is guaranteed;* (3)

if $R_0(i)$ is satisfied for every processor p_i and R_2 is satisfied, then $Bounded\,Waiting$ is guaranteed with waiting time bounded by TM_{phase}.

Proof. First we notice that if $R_0(i)$ and $R_1(i)$ are $true$, then p_i is added to $ring_0$ or new_set_0 in infinitely many phases for the following reason: If it is not true, p_i stops being visited by tokens eventually and $timer_i^{join}$ will not be reset to a non-zero value, so infinitely many $send_i^j$ events are activated, in which infinitely many $j\text{-}token$s are sent to p_0, thus infinitely many $j\text{-}token$s arrive at p_0 by R_0 and p_i is added to new_set_0 in infinitely many phases, which is a contradiction.

Note in any phase such that $p_i \in new\_set_0$, at least one token that includes p_i in new_set is generated during $[T, T+TM_{phase}-M_{lt}]$, where T is the time when this phase starts. From the discussion above, we know there are infinitely many phases such that $p_i \in new\_set_0$, so we have $p_i \in ring_0$ in infinitely many phases by $R_1(i)$ and Lemma 19. Note in each of the infinitely many phases such that $p_i \in ring^0$, at least one token that includes $p_i \in ring$ is generated during $[T, T+TM_{phase}-M_{lt}]$, where T is the time the phase starts. So p_i enters the critical section infinitely often by Lemma 19 and $R_1(i)$. Thus (1) and (2) are proved.

Now we consider $Bounded\ Waiting$. In each phase, p_0 keeps generating tokens. By R_2, at least one token which is generated within time $TM_{phase} - M_{lt}$ from the beginning of this phase visits all the members and comes back to p_0 within L hops, so every processor enters the critical section in this phase by Lemma 19. By Lemma 3, the waiting time is bounded by TM_{phase}. ■

9 Conclusion

In this paper we presented a self-stabilizing mutual exclusion algorithm for mobile ad hoc networks, which guarantees safety properties under arbitrary mobility, and guarantees different levels of progress under different levels of constraints. Theorem 15 implies that the time for the algorithm to converge is $O\left(L^2\right)$ and the waiting time implied by Theorem 20 is $O(L)$.

The assumption of a distinguished processor can be relaxed by using a self-stabilizing leader election algorithm for mobile ad hoc networks. ([18] is a first step toward the development of such an algorithm, in which a "weak" self-stabilizing algorithm is presented.)

The assumption that a processor stays in the critical section only for a negligible time can be relaxed by replacing d by $d + C$ in the definition of M_{lt} in Fig 3, where C is the maximum time a processor stays in the critical section. No change is needed in the analysis, since the analysis is done in terms of M_{lt}.

Simulation results in [13] [12] show that the (unbounded) LRV algorithm performs well in mobile networks. In particular, the round length is very close to the optimal value n, where round length is the number of hops for a token to be forwarded to all the nodes and n is the number of nodes in the network. If the parameter L to our bounded version of LRV is chosen to be on the order of the "usual" round length, then the conditions for progress (R_0, R_1, and R_2) are likely to be satisfied. With such a value of L, the simulation results indicate that $O\left(n^2\right)$ and $O(n)$ are reasonable approximations to the convergence time and waiting time, respectively. Thus we can see that the requirements on LRV's performance to guarantee different levels of progress are feasible in practice.

In our future work, we are interested in characterizing the specific mobility patterns in which the progress property can be guaranteed. More practically, we would like to evaluate the usefulness of the heuristic by which the predefined ring is updated and compare it to others.

References

[1] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proceedings of the 7th IEEE Symposium on Computers and Communications (ISCC 2002)*, 2002.

[2] C. Basile, M.-O. Killijian, and D. Powell. A survey of dependability issues in mobile wireless networks, http://citeseer.nj.nec.com/basile03survey.html.

[3] Y. Chen and J. L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In *the 6th Annual International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM'2002)*, 2002.

[4] A. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *5th International Colloquium on Structural Information and Communication Complexity (SIROCCO'98)*, 1998.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[6] S. Dolev. Optimal time self-stabilization in uniform dynamic systems. *Parallel Processing Letters*, 8(1):7–18, 1993.

[7] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[8] S. Dolev, E. Schiller, and J. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *21st IEEE Symposium on Reliable Distributed Systems*. Oct, 2002.

[9] K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, and R. B. Tan. Fundamental control algorithms in mobile networks. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 251–260, 1999.

[10] T. Herman. Self-stabilization bibliography: Access guide. *Chicago Journal of Theoretical Computer Science*, November 1996.

[11] M. Lauer and M. Matthes. Mobicom poster: Elan—an e-learning infrastructure for ad-hoc networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(1):53–55, 2003.

[12] N. Malpani, Y. Chen, N. Vaidya, and J. Welch. Distributed token circulation in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, to appear.

[13] N. Malpani, N. Vaidya, and J. Welch. Distributed token circulation in mobile ad hoc networks. In *Proc. 9th International Conference on Network Protocols(ICNP)*, Nov. 2001.

[14] N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proc. Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, 2000.

[15] F. Petit and V. Villain. Color optimal self-stabilizing depth-first token circulation for asynchronous message-passing systems. In *Proc. of the ISCA 10th Conf. Parallel and Distributed Computing Systems*, Oct, 1997.

[16] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[17] G. Varghese. Self-stabilization by counter flushing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994.

[18] S. Vasudevan, N. Immerman, J. Kurose, and D. Towsley. A leader election algorithm for mobile ad hoc networks. Technical Report 0301, Department of Computer Science, University of Massachusetts, January 13, 2002.

[19] J. Walter, J. Welch, and N. Vaidya. Mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 9(6):585–600, Nov. 2001.