

# Self-Stabilizing Mutual Exclusion Using Tokens in Mobile Ad Hoc Networks

Yu Chen      and      Jennifer L. Welch  
Department of Computer Science  
Texas A&M University  
College Station, TX 77843-3112, U.S.A.  
{ychen, welch}@cs.tamu.edu

## Abstract

*In this paper, we propose a self-stabilizing mutual exclusion algorithm using tokens for mobile ad hoc networks. Our algorithm is based on dynamic virtual rings formed by circulating tokens. We require the topology to be static while the algorithm is converging. But after it has converged, under a restricted mobility assumption, it guarantees both the safety and liveness properties of mutual exclusion; under arbitrary mobility, it cannot guarantee the liveness property, but it does guarantee the safety property.*

## 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 as pointed out in [7], much of the work in this area has focused on routing and medium access control protocols, and there is less work on distributed services. Past work on distributed services focused on non-fault-tolerant algorithms. For example, in [7], several distributed token circulation algorithms for mobile ad hoc networks are studied, one of which, the  $\mathcal{LRV}$  algorithm, shows quite good performance in simulations. And in [10], a mutual exclusion algorithm for mobile ad hoc networks is presented. In this paper, we propose a self-stabilizing mutual exclusion algorithm for mobile ad hoc networks, denoted by  $ss\mathcal{ME}$ , which is based on the  $\mathcal{LRV}$  algorithm.

The concept of self-stabilization was first introduced by Dijkstra in [4], in which a system is defined as self-stabilizing if, starting from an arbitrary configuration, it is guaranteed to converge to a "legitimate" configuration in finite time. The execution starting from that legitimate configuration shows desired behavior. Self-stabilization is an important technique for designing algorithms that tolerate arbitrary transient faults, and a lot of work has been done on it for static networks. However there is very little work on self-stabilization for mobile ad hoc networks (one exception is [6]). It is inefficient to apply a self-stabilizing algorithm for static networks directly in mobile ad hoc networks for the following reason. In these algorithms, a topology change is generally considered as a kind of transient fault and the algorithms usually require the topology to be static for converging to and staying in legitimate configurations. If such an algorithm is applied in a mobile ad hoc network, which typically experiences frequent topology changes, it would try to converge every time a topology change occurs, which is quite inefficient and might cause the algorithm never to reach a legitimate configuration.

In this paper, we focus on the mutual exclusion problem in mobile ad hoc networks and propose an algorithm  $ss\mathcal{ME}$  which can tolerate restricted topology changes after it has converged. A mutual exclusion algorithm should satisfy two properties: *safety* — there is no more than one processor in the critical section; and *liveness* — every processor enters the critical section infinitely often. The mutual exclusion problem 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 safety and liveness properties (no more than one token in the system and every processor gets it infinitely often), clearly this simple approach to solving mutual exclusion is correct.

Our algorithm is based on a token circulation algorithm called  $\mathcal{LRV}$  [7], 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 is in the critical section in such a way that only one processor is in the critical section even when there are multiple tokens in the system. Since in an adversarial mobility pattern, a processor might always avoid the token holder and never get a token, certain constraints on the mobility are required to guarantee the liveness property. In this paper, we define constraints on mobility that suffice to ensure the liveness property of mutual exclusion.

Self-stabilizing mutual exclusion and token circulation algorithms have been designed for static networks. However, after they converge, they do not guarantee their safety properties in the presence of ongoing topology changes; thus a mutual exclusion algorithm based on one of these token circulation algorithms in the simple way mentioned above cannot guarantee its safety property. In [4] a self-stabilizing algorithm for mutual exclusion on a ring is presented. Self-stabilizing depth-first token circulation algorithms are presented in [8] and [3]. If they are applied in mobile ad hoc networks, after they have converged, mobility might break the safety property of mutual exclusion or token circulation. In this paper,  $ss\mathcal{ME}$  can guarantee the safety property in the presence of arbitrary mobility after it has converged.

We require the network to be static while  $ss\mathcal{ME}$  is converging. But after it has converged, under the given mobility constraints,  $ss\mathcal{ME}$  guarantees both the safety and liveness properties; under arbitrary mobility pattern it does not guarantee the liveness property but it does guarantee the safety property. This paper draws on ideas from [4] and [9]. In [4], a counter is used at each processor for mutual exclusion on a ring. In [9], the idea of “counter flushing” is applied on a ring, a tree and a general graph with a spanning tree. Here we apply this idea to the dynamic virtual rings formed by circulating tokens in a mobile ad hoc network with general topology. If we apply the idea in [4] on a fixed virtual ring for mobile ad hoc networks, we can get a self-stabilizing algorithm if we put some constraints on the mobility to ensure the liveness property. But a fixed virtual ring is quite inefficient in mobile ad hoc networks. Thus dynamic virtual rings are used in  $ss\mathcal{ME}$  to reflect the changing topology.

## 2 System Model

In this paper the 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 have the following assumptions for the mobile ad hoc network  $G$ :

- $G$  is a message passing system with an upper bound  $d$  on the point-to-point message delay.
- There is an upper bound  $g$  on the number of messages generated by a processor in each time unit.
- The messages in transit are in a FIFO incoming queue at the receiving processor.
- The mobile processors in  $G$  are always strongly connected and each processor knows the ids of the other processors.

In this paper we assume there is a distinguished processor in  $G$  (later we will discuss relaxing this assumption). In the sequel we denote the set of all processors in the system by  $P$ , the distinguished processor by  $p_0$ , and the other processors by  $p_i, i \in \{1, \dots, N - 1\}$ , where  $N$  is the number of processors in  $G$ . For simplicity, we assume the id of  $p_i$  is  $i$ .

We assume there is a reliable underlying communication service so that we are not concerned about message loss. This communication service on processor  $p_i$  provides the following API function for point-to-point communication between adjacent processors:

- $Send(id\ j, message\ msg)$ : Forwards the message  $msg$  to the incoming FIFO queue at  $p_j$ , where  $p_j$  is a neighbor of  $p_i$ .

Given a mobile ad hoc network  $G$ , 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 status of the local variables at  $p_i$ . We denote the incoming queue at  $p_i$  by  $p_i.queue$ , and the state of  $p_i.queue$  by  $qstate_i$ , which consists of the sequence of messages in  $p_i.queue$ . Thus the *configuration* of  $G$  can be described as  $\{ \langle pstate_0, \dots, pstate_{n-1} \rangle, \langle qstate_0, \dots, qstate_{n-1} \rangle, \tau \}$ , where  $\tau$  is 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*. Each event is associated with a real number, indicating the real time of its occurrence. A local computation event is initiated by the receipt of a message or a timeout expiring. A local computation event can change the state of the processor at which it occurs, and enqueue messages in the incoming queues of neighboring processors. A topology change event changes the topology  $\tau$  (specific restrictions on the topology changes are discussed below). We assume the local computation time is negligible.

An *execution* of an algorithm in  $G$  is an alternating sequence of configurations and events  $c_0, e_1, c_1, e_2, \dots$  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 sent and received and topology changes occur in ways that are consistent with the assumptions listed above and the mobility constraints discussed below. (3) Each configuration after the first follows appropriately from the previous, according to the specification of the intervening event.

### 3 Task of Mutual Exclusion

We now define the *mutual exclusion task*, similarly to [4]. In the definition, “processor  $p_i$  being in the critical section in configuration  $c$ ” refers to a predicate whose specifics will depend on the particular algorithm.

**Definition 1 (Task of Mutual Exclusion  $\mathcal{ME}$ ).** We define the task of mutual exclusion by a set of executions, denoted by  $\mathcal{ME}$ , such that the following conditions hold:

- *safety*: There is no more than one processor in the critical section in any configuration; and
- *liveness*: Every processor enters the critical section infinitely often in each infinite execution.

In this paper we focus on a self-stabilizing  $\mathcal{ME}$  algorithm for mobile ad hoc networks with certain constraints on the mobility. The definition of a *self-stabilizing  $\mathcal{ME}$  algorithm* in a mobile ad hoc network is defined as follows:

**Definition 2 (Self-Stabilization).** An algorithm is self-stabilizing for  $\mathcal{ME}$  with respect to a mobility pattern in a mobile ad hoc network if, starting from an arbitrary configuration, every infinite execution of the algorithm with this mobility pattern has a suffix that belongs to  $\mathcal{ME}$ .

Similarly to [5], we define a *safe configuration* for  $\mathcal{ME}$  with respect to an algorithm and a mobility pattern in a mobile ad hoc network:

**Definition 3 (Safe Configuration).** *A configuration  $c$  is safe for  $\mathcal{ME}$  with respect to an algorithm and a mobility pattern if every infinite execution of the algorithm with that mobility pattern that starts from  $c$  belongs to  $\mathcal{ME}$ .*

We say an algorithm is *stabilized* or *has converged* if it has entered a safe configuration.

#### 4. Self-Stabilizing Mutual Execution Algorithm $ss\mathcal{ME}$

$ss\mathcal{ME}$  is based on circulating token messages. The distinguished processor  $p_0$  keeps generating tokens and each token is routed independently by  $\mathcal{LRV}$ . When a processor  $p$  receives a token, it checks its local state and the token's state to decide whether it is in the critical section. After  $ss\mathcal{ME}$  has stabilized, there might be multiple tokens in the system, but no more than one of them can enable a processor to be in the critical section.

In this paper we define a mobility pattern, called “ $\mathcal{LRV}$ -friendly” (defined in Section 4.1), under which the liveness property of  $\mathcal{ME}$  can be guaranteed. We require the network to be static until  $ss\mathcal{ME}$  enters a safe configuration. Once  $ss\mathcal{ME}$  is stabilized, in  $\mathcal{LRV}$ -friendly mobile ad hoc networks, it guarantees both the safety and liveness properties of  $\mathcal{ME}$ ; otherwise it cannot guarantee the liveness property but it does guarantee the safety property. Actually any token circulation algorithm can be used here as long as it satisfies Property 1 (defined in Section 4.1) and works in a “friendly” (defined in Section 4.1) mobile ad hoc network.

##### 4.1 $\mathcal{LRV}$ algorithm

The non-self-stabilizing  $\mathcal{LRV}$  algorithm is introduced in [7], in which there is a unique token  $t$  that contains the data structure  $t.ts[1..N]$ , where  $t.ts$  is an array of  $N$  integers (the “timestamp array”) initialized to all 0's and  $t.ts[i]$  is a timestamp indicating when the processor received the token most recently.

$\mathcal{LRV}$  code for  $p_i, 0 \leq i < N$ :

Event  $Receive_i(\text{token } t)$ :

- 1  $t.ts[i] = \max(t.ts) + 1$ ;
- 2  $next = \text{index of } t.ts \text{ entry among all neighbors of } p_i \text{ with minimum value (break ties by id)}$ ;
- 3 send  $t$  to  $p_{next}$ ;

Given an execution  $e$  of  $\mathcal{LRV}$ , let the *token holder list*  $th(e)$  be the sequence of ids indicating the order in which processors have held the token. We partition  $th(e)$  into *rounds* as follows: round 0 is the minimal prefix of  $th(e)$  in which every processor id appears at least once; round  $i, i > 0$ , is the minimal prefix of the suffix of  $th(e)$  following round  $i - 1$  in which every processor id appears at least once. The *length* of a round (resp. a token holder list) is the number of ids in the round (resp. token holder list).

From the above code, we notice that the choice of the next token holder and the update to  $t.ts$  are determined by the current token holder  $p$ , the neighbor set of  $p$  and the current state of  $t$ . So we have the following property of  $\mathcal{LRV}$  in a static network:

**Property 1** *If two configurations have the same topology  $\tau$ , the same token state, and the same token holder, then the executions of  $\mathcal{LRV}$  from those configurations, in which  $\tau$  never changes, have the same token holder list.*

In [1], it is shown that there exist (static) graphs that have executions of  $\mathcal{LRV}$  with exponential round length and that an upper bound on the round length of  $\mathcal{LRV}$  for any static graph is  $O(N \cdot \Delta^D)$ , where  $\Delta$  is the degree and  $D$  is the diameter of the graph. However, in the simulation results in [7] the observed round lengths were very small,

in fact, close to  $N$ , for both static and dynamic networks. Thus, we are interested in practical situations when the round length can be estimated as something significantly smaller than  $O(N \cdot \Delta^D)$  and the mobility “cooperates” with this estimate. We denote this estimated round length as  $RL_{est}$ .

**Definition 4 ( $\mathcal{LRV}$ -friendly).** A mobility pattern is  $\mathcal{LRV}$ -friendly with respect to  $RL_{est}$  if, in every execution satisfying the pattern, whenever infinitely many tokens are generated and forwarded by  $\mathcal{LRV}$ , infinitely many of them finish  $N$  consecutive rounds with total length no more than  $N \cdot RL_{est}$ .

We require  $RL_{est}$  to be such that a static network is  $\mathcal{LRV}$ -friendly. Note that by [1] as mentioned above, there is always some value for  $RL_{est}$  that will work.

In  $ss\mathcal{ME}$  we use a variant of  $\mathcal{LRV}$ , denoted by  $\mathcal{LRV}(Max_{ts})$  where  $Max_{ts} = N \cdot RL_{est}$ , which is similar to  $\mathcal{LRV}$  except that the timestamps in  $\mathcal{LRV}(Max_{ts})$  are in the range  $[0, Max_{ts}]$  and the operation “+” on  $t.ts$  is addition mod  $Max_{ts} + 1$ . Property 1 is still true in  $\mathcal{LRV}(Max_{ts})$  and we have the following lemma:

**Lemma 1** In every execution whose mobility pattern is  $\mathcal{LRV}$ -friendly with respect to  $RL_{est}$ , whenever infinitely many tokens are generated and forwarded by  $\mathcal{LRV}(Max_{ts})$ , infinitely many of them finish  $N$  consecutive rounds with total length no more than  $N \cdot RL_{est}$ .

Due to space limitations, we do not provide the proof in this paper; it is in the full paper [2].

## 4.2 Algorithm $ss\mathcal{ME}$

Besides the timestamp array  $t.ts$ , each token  $t$  contains the following data structures, where  $Max_{ts} = N \cdot RL_{est}$  and  $Max_{tid} = g \cdot N \cdot RL_{est} \cdot d + 2 = g \cdot d \cdot Max_{ts} + 2$ .  $Max_{ts}$  is the maximum timestamp of a token generated by  $p_0$  before it finishes  $N$  rounds.  $Max_{tid}$  is two greater than the maximum number of tokens generated by  $p_0$  during this time.

- $id$ : integer in  $[0, Max_{tid}]$ , the id of token  $t$ .
- $ts[0..N]$ : array of integers, each value in  $[0, Max_{ts}]$ , the timestamp array of token  $t$ .
- $predefinedPath$ : permutation of  $\{0, \dots, N\}$ , list of processor ids indicating the order in which they are to be visited by token  $t$ .
- $visitedSet$ : subset of  $\{0, \dots, N\}$ , indicating processors already visited by token  $t$ .
- $routingOrder$ : permutation of any subset of  $\{0, \dots, N\}$ , list of processor ids indicating the order in which token has been routed, which will be used to update  $predefinedPath$ , so that it will adapt to reality.

Each processor  $p_i$ ,  $0 \leq i < N$ , has in its local state the following variables:

- $tid_i$ : integer in  $[0, Max_{tid}]$ , the local token id at  $p_i$ ;

Processor  $p_0$  also has the following variable:

- $predefinedPath_0$ : permutation of  $\{0, \dots, N\}$ , list of processor ids which is updated by  $t.routingOrder$  and is used as a token’s  $predefinedPath$  when the token is generated.

Since the self-stabilizing algorithm might start in an arbitrary configuration, the data in the local variables or in the tokens might be out of range. If such data is detected,  $ss\mathcal{M}\mathcal{E}$  sets the variable to any predefined value. Thus in the sequel we assume all data is in range.

In  $ss\mathcal{M}\mathcal{E}$ ,  $p_0$  keeps sending tokens with its local token id,  $tid_0$ , in event  $Timeout_0$ , which handles the case that all tokens are lost due to faults. By our predicate for the critical section (defined below), the algorithm guarantees that only one token can enable a processor to be in the critical section even when there are multiple tokens, so the timeout value of the timer doesn't matter.

Each token  $t$  is routed by  $\mathcal{LRV}(Max_{ts})$  in function  $NextID(t)$ , but it visits the processors according to  $t.predefinedPath$ . Each time processor  $p_i$  receives  $t$ , it checks whether all the processors previous to it in  $t.predefinedPath$  have been visited by  $t$ , that is, are in  $t.visitedSet$ . If not,  $t$  is routed by  $p_i$  to the next token holder decided by  $NextID(t)$ , otherwise we say  $p_i$  is visited by  $t$  and  $p_i$  is added to  $t.visitedSet$ . When  $p_i$  is visited by  $t$ ,  $p_i$  checks  $tid_i$  and  $t.id$  to decide whether it is in the critical section before  $p_i$  routes  $t$  to the next token holder.

In  $ss\mathcal{M}\mathcal{E}$  the predicate for the critical section is defined on configuration  $c$  and processor id  $i$  as follows: if  $i \neq 0$ ,  $pred(c, i)$  is (all the processors previous to  $p_i$  in  $t.predefinedPath \subseteq t.visitedSet$ )  $\&\&(t.id == (tid_i + 1) \bmod (Max_{tid} + 1))$ , otherwise  $pred(c, i)$  is  $(P == t.visitedSet) \&\&(t.id == tid_0)$ , where  $t$  is the first token in  $p_i.queue$ .

$ss\mathcal{M}\mathcal{E}$  code of  $p_i, i \neq 0$ :

Event  $Receive_i(\text{token } t)$ :

Precondition: token  $t$  is the first token in  $p_i.queue$ .

Action:

```

01   remove  $t$  from  $p_i.queue$ ;

02   if ( $max(t.ts) \geq Max_{ts}$ ) then
03       return; /* token  $t$  is discarded */
04   endif;

05   if ( {all the processors previous to  $p_i$  in  $t.predefinedPath$ }  $\subseteq t.visitedSet$  ) then
       /*  $p_i$  is visited by  $t$  */
06       add  $p_i$  to  $t.visitedSet$ ;
07       if ( $t.id == (tid_i + 1) \bmod (Max_{tid} + 1)$ ) then
           /* Critical Section */
08            $tid_i = t.id$ ;
09       endif
10   endif

       /* the token is routed to the next token holder */
11   if (  $p_i \notin t.routingOrder$  ) then
12       append  $p_i$  to the end of  $t.routingOrder$ ;
13   endif
14    $nextID = NextID(t)$ ;
15    $Send(nextID, t)$ ;

```

$ss\mathcal{M}\mathcal{E}$  code of  $p_0$ :

Event  $Timeout_0$ :

Precondition: timeout.

Action:

```
16   $t = GenerateToken(tid_0, predefinedPath_0)$ ;  
17   $nextID = NextID(t)$ ;  
18   $Send(nextID, t)$ ;  
19  reset timer;
```

Event  $Receive_0(token\ t)$ :

Precondition:  $t$  is the first token in  $p_0.queue$ .

Action:

```
20  remove  $t$  from  $p_0.queue$ ;  
  
21  if ( $max(t.ts) \geq Max_{ts}$ ) then  
22      return; /* token  $t$  is discarded */  
23  endif;  
  
24  if ( $P == t.visitedSet$ ) then  
25      if ( $t.id == tid_0$ ) then  
26          /* a new phase is started */  
27           $tid_0 = (tid_0 + 1) \bmod (Max_{tid} + 1)$ ;  
28           $predefinedPath_0 = t.routingOrder$ ;  
29          /* Critical Section */  
30          /* token  $t$  is discarded and a new token  $t'$  is generated */  
31           $t' = GenerateToken(tid_0, predefinedPath_0)$ ;  
32           $nextID = NextID(t')$ ;  
33           $Send(nextID, t')$ ;  
34      else  
35          /* token  $t$  is discarded */  
36      endif  
37  else /* not every processor is visited */  
38       $nextID = NextID(t)$ ;  
39       $Send(nextID, t)$ ;  
40  endif
```

*token*  $GenerateToken(tokenID\ tid, processorList\ pPath)$ :

```
37  create new token data structure  $t$ ;  
38   $t.id = tid$ ;  
39   $t.predefinedPath = pPath$ ;  
40   $t.routingOrder = \langle p_0 \rangle$ ;  
41   $t.visitedSet = \{p_0\}$ ;  
42   $t.ts = [0, 0, \dots, 0]$ ;  
43  return  $t$ ;
```

*ssME* code of  $p_i, 0 \leq i \leq N - 1$ :

*id*  $NextID(token\ t)$ :

```
44   $t.ts[i] = (max(t.ts) + 1) \bmod (Max_{ts} + 1)$ ;
```

45      $next = \text{index of } t.ts \text{ entry among all neighbors of } p_i \text{ with minimum value (break ties by id);}$   
46     return  $next$ ;

## 5. Self-Stabilization of $ss\mathcal{ME}$

In this section, we consider the self-stabilizing property of  $ss\mathcal{ME}$ . We first discuss some basic properties of  $ss\mathcal{ME}$  in Lemma 2, 3, 4 and 5. Then we focus on the configurations with property  $\mathcal{SC}$  (defined below), which have some interesting properties described in Lemma 7. Lemma 6 shows that starting from an arbitrary configuraion,  $ss\mathcal{ME}$  will eventually enter a configuration with property  $\mathcal{SC}$  if the topology remains static. Lemma 8 shows that the configurations with property  $\mathcal{SC}$  are safe configurations. We conclude in Theorem 9. Due to space limitations, we do not provide the proofs in this paper; they are in the full paper [2].

By the code of  $ss\mathcal{ME}$ , each time processor  $p_i$  receives a token  $t$ ,  $t$  is removed from  $p_i.queue$ , after which  $t$  might or might not be routed to the next token holder. If  $t$  is not routed to the next token holder, we say  $t$  is *discarded* by processor  $p_i$ . There are two situations in which  $t$  is discarded. One is in line 3 and line 22 if  $max(ts) \geq Max_{ts}$ , another is in lines 25-32 with line 24 *true*. Here we define the *lifetime* of a token  $t$  as the time interval between the time when  $t$  is discarded and the time of initial configuration if  $t$  exists initially; otherwise we define the *lifetime* as the time interval between the time when  $t$  is discarded and the time when  $t$  is generated. By lines 2-4 and 21-23, we have the following property about token's lifetime:

**Lemma 2** *Starting from an arbitrary configuration, every token's token holder list is no longer than  $Max_{ts}$  and its lifetime is no more than  $Max_{ts} \cdot d$ .*

By the requirement of  $RL_{est}$  for the static network, we have the following lemma:

**Lemma 3** *In a static network, every token generated by  $p_0$  will arrive at  $p_0$  with line 24 *true* eventually.*

We define the *phases* of an execution of  $ss\mathcal{ME}$  as follows:

**Definition 5 (Phase).** *Given an execution  $\sigma$  of  $ss\mathcal{ME}$ , we define the phases of  $\sigma$  as follows: phase 0 starts at the beginning of  $\sigma$ , and ends just before the first event of  $Receive_0(t)$  with lines 24 and 25 *true* in  $\sigma$ ; phase  $i$ ,  $i > 0$ , starts immediately after the end of phase  $i - 1$  and ends just before the second occurrence after phase  $i - 1$  of  $Receive_0(t)$  with lines 24 and 25 *true*.*

By the definition of phases and lines 26 and 27 of the code, we have the following property about phases of  $ss\mathcal{ME}$ .

**Lemma 4** *The tokens generated by  $p_0$  in one phase have the same id and predefinedPath, and the token id of tokens generated in the next phase is one larger (mod ( $Max_{tid} + 1$ )) than that of the tokens generated in the previous phase.*

By the definition of  $\mathcal{LRV}$ -friendly and Lemma 1, we have the following lemma:

**Lemma 5** *Starting from an arbitrary configuration, there are infinitely many phases in an  $\mathcal{LRV}$ -friendly execution of  $ss\mathcal{ME}$ .*

Now we consider the configurations of  $ss\mathcal{ME}$  with the following property, denoted by  $\mathcal{SC}$ :

$\mathcal{SC}$ : There is a common token id  $l$ , such that all the tokens have the same token id  $l$  and each processor  $p_i$ 's local token id  $tid_i$  is  $l$ .

Now we show that starting from an arbitrary configuration,  $ss\mathcal{ME}$  will eventually enter a configuration with property  $\mathcal{SC}$  if the network remains static. By Lemma 2, the tokens in the initial configuration will be discarded in time  $Max_{ts} \cdot d$ , after which the tokens travel in a virtual FIFO pipe by Property 1 of  $\mathcal{LRV}(Max_{ts})$  if the topology remains static. The local token id on each processor is updated when it is visited by a token. We can prove that after  $(Max_{tid} + 1)$  phases, all the processors have the same local token ids, and at the end of this phase, all the tokens have the same token id as the local token ids of the processors.

**Lemma 6** *Starting from an arbitrary configuration, algorithm  $ss\mathcal{ME}$  will enter a configuration with property  $\mathcal{SC}$  in time  $(Max_{tid} + 2) \cdot Max_{ts} \cdot d$  if the network remains static, where  $Max_{ts} = N \cdot RL_{est}$  and  $Max_{tid} = g \cdot d \cdot Max_{ts} + 2$ .*

We have the following properties for the configuration with property  $\mathcal{SC}$ .

**Lemma 7** *Given an execution of  $ss\mathcal{ME}$  with  $k + 1$  phases starting from a configuration satisfying property  $\mathcal{SC}$  with common token id  $l$ , we have the following properties:*

- *In phase  $k$ , there is no token with id of  $(l + k + 1)$  or  $(l + k + 2)$ .*
- *All the tokens in the system with the same id are generated in the same phase.*
- *At the end of phase  $k$ , all the processors have the same local token id of  $l + k$ .*

Consider the execution starting from a configuration satisfying property  $\mathcal{SC}$  with common id  $l$ . By our predicate,  $p_0$  is in the critical section at the end of every phase and no other processors are in the critical section in phase 0. Since each other processor enters the critical section only when it is visited by the first token with id  $l + k$  in phase  $k, k > 0$ , by the predicate, the safety property can be proved by Lemma 7. Since each processor enters the critical section in each subsequent phase and there are infinitely many phases in an  $\mathcal{LRV}$ -friendly network by Lemma 5, the liveness property of  $\mathcal{ME}$  can be proved. So we have the following lemma:

**Lemma 8** *The configurations satisfying property  $\mathcal{SC}$  are safe configurations of  $\mathcal{ME}$  with respect to  $ss\mathcal{ME}$  and an  $\mathcal{LRV}$ -friendly mobility pattern.*

Now we discuss what kind of result we can get for  $ss\mathcal{ME}$  in a non- $\mathcal{LRV}$ -friendly mobile ad hoc work. Lemmas 6–8 rely on Lemmas 2–5, of which Lemma 5 is the only one based on the property of “ $\mathcal{LRV}$ -friendly”. In Lemma 6, we only consider the static network. Lemma 7 and the proof of the safety property in Lemma 8 do not rely on Lemma 5, that is, they do not rely on the property of “ $\mathcal{LRV}$ -friendly”. However the proof of the liveness property of Lemma 8 does rely on this property, since in a non- $\mathcal{LRV}$ -friendly mobile ad hoc network, there might not be infinitely many phases in an execution of  $ss\mathcal{ME}$ , and the processors might only enter the critical section finitely often.

**Theorem 9** *We have the following conclusion about the self-stabilization property of  $ss\mathcal{ME}$ :*

- *Starting from an arbitrary configuration,  $ss\mathcal{ME}$  enters a configuration satisfying property  $\mathcal{SC}$  in time  $(Max_{tid} + 2) \cdot Max_{ts} \cdot d$  if the network remains static, where  $Max_{ts} = N \cdot RL_{est}$  and  $Max_{tid} = g \cdot d \cdot Max_{ts} + 2$ .*
- *In an  $\mathcal{LRV}$ -friendly mobile ad hoc network,  $ss\mathcal{ME}$  guarantees both the safety and liveness properties of  $\mathcal{ME}$  in any execution starting from a configuration satisfying property  $\mathcal{SC}$ .*
- *In a non- $\mathcal{LRV}$ -friendly mobile ad hoc network,  $ss\mathcal{ME}$  cannot guarantee the liveness property of  $\mathcal{ME}$ , but it does guarantee the safety property in any execution starting from a configuration satisfying property  $\mathcal{SC}$ .*

## 6 Conclusion

In this paper we presented a self-stabilizing mutual exclusion algorithm for mobile ad hoc networks. We require the topology to be static while the algorithm is converging. Once it is stabilized, it guarantees both the safety and liveness properties of mutual exclusion in an  $\mathcal{LRV}$ -friendly network; otherwise, it cannot guarantee the liveness property but it does guarantee the safety property. The assumption of a distinguished processor can be relaxed by an existing or future self-stabilizing leader election algorithm for mobile ad hoc networks. In our future work, we are interested in characterizing the specific mobility patterns in which the liveness property can be guaranteed. We are also interested in relaxing the requirement of a static topology during the stabilization of the algorithm. More practically, we would like to evaluate the usefulness of the heuristic by which the predefined path is updated and compare it to others.

**Acknowledgments:** We thank Evelyn Tumlin Pierce and Elad Schiller for helpful discussions.

## References

- [1] Y. Chen and J. Welch. Bounds on Round Length for the Least-Recently-Visited Token Circulation Algorithm. Technical Report 2002-4-1, Department of Computer Science, Texas A&M University, April 2002. url: <http://www.cs.tamu.edu/people/ychen/Research/Paper/lrv.pdf>
- [2] Y. Chen, and J. Welch. Self-Stabilizing Mutual Exclusion Using Tokens in Mobile Ad Hoc Networks. Technical Report 2002-4-2, Department of Computer Science, Texas A&M University, April 2002. url: [http://www.cs.tamu.edu/people/ychen/Research/Paper/sstcSM\\_lrv.pdf](http://www.cs.tamu.edu/people/ychen/Research/Paper/sstcSM_lrv.pdf)
- [3] A.K. Datta, C. Johnen, F. Petit, and V. Villain. Self-Stabilizing Depth-First Token Circulation In Arbitrary Rooted Networks. In *SIROCCO'98, The 5th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 229-243, 1998.
- [4] E. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643-644, 1974.
- [5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
- [6] S. Dolev, E. Schiller, and J. Welch. Brief Announcement: Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks. *ACM Symposium on Principles of Distributed Computing*, July 2002, to appear.
- [7] N. Malpani, N. Vaidya and J. Welch. Distributed Token Circulation in Mobile Ad Hoc Networks. In *Proc. 9th International Conference on Network Protocols (ICNP)*, November 2001.
- [8] 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*, New Orleans, Oct. 1997
- [9] G. Varghese. Self-Stabilization by counter flushing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, August 1994.
- [10] J. Walter, J. Welch, and N. Vaidya. Mutual Exclusion Algorithm for Ad Hoc Mobile Networks. *Wireless Networks*, Vol. 9, No. 6, November 2001, pages 585-600.