Efficient and Robust Local Mutual Exclusion in Mobile Ad Hoc Networks

(Extended Abstract)

Hagit Attiya
Dept. of Computer Science
Technion, Israel
hagit@cs.technion.ac.il

Alex Kogan
Dept. of Computer Science
Technion, Israel
sakogan@cs.technion.ac.il

Jennifer L. Welch*
Dept. of Computer Science
Texas A&M University, TX
welch@cs.tamu.edu

Abstract

This paper presents two algorithms for the local mutual exclusion problem, an extension of the dining philosophers problem for mobile ad hoc networks. A solution to this problem allows nodes that are currently geographically close to obtain exclusive access to a resource. The algorithms exhibit different tradeoffs between response time and failure locality (the size of the neighborhood adversely affected by a node crash).

The first algorithm has two variations, one of which has response time that depends very weakly on the number of nodes in the entire system and is polynomial in the maximum number of neighboring nodes; the failure locality, although not optimal, is small and grows very slowly with system size. The second algorithm has optimal failure locality and response time that is quadratic in the number of nodes. A pleasing aspect of this algorithm is that, when run in a system with no node movement, it has linear response time, improving on previous results for static algorithms with optimal failure locality.

1 Introduction

Advances in wireless technology have made it possible to deploy mobile ad hoc networks (MANETs), collections of moving computing entities (nodes) that communicate through wireless transmission, usually without the assistance of a fixed infrastructure. Commonly suggested applications for MANETs include emergency situations, military missions, and environmental data collection. Although many hardware challenges have been solved, programming applications for MANETs remains a challenge. One of the complications is that the communication topology changes unpredictably as nodes move. The availability of tried-and-tested primitives, or building blocks, for common problems in MANETs should help in developing applications.

Local mutual exclusion (LME) is a primitive that we believe will help in various applications. Consider the situation where each node has a certain part of its code, called its *critical section*, that needs to be executed every now and then in exclusion with respect to the neighbors with whom it can communicate directly (roughly speaking, nodes that are physically nearby). We seek an algorithm to ensure that no two neighboring nodes are in their critical sections simultaneously. Furthermore, the algorithm should ensure that, under some reasonable conditions, whenever a node wants to enter its critical section, it eventually is allowed to do so.

In the (global) mutual exclusion problem, no two nodes in the system, no matter how far apart, should be in their critical sections simultaneously. Although there has been previous work solving this problem in MANETs (e.g., [1, 11, 14, 16]), it appears to have fewer potential applications than the problem studied in this paper.

A local mutual exclusion algorithm can be used to facilitate communication in a MANET. For instance, nearby nodes can compete for exclusive access to a dedicated wireless channel or to a satellite uplink facility using this algorithm. They will be ensured of all eventually getting a turn to use the communication channel exclusively. Another application of local mutual exclusion is to arbitrate access to some piece of specialized hardware in a region, such as a more powerful computer in the system (e.g., a repository for collected data [11]) or the computer that is running a projector in a meeting room.

Nodes in a MANET are subject to harsh environment conditions and often have limited, and irreplaceable, battery life. Consequently, they can fail, and it is important to develop fault-tolerant solutions. One criterion for measuring solutions to the local mutual exclusion problem is *failure locality*, defined originally in [4], which measures

*The work of this author was supported in part by NSF grant 0500265 and Texas Higher Education Coordinating Board grant ARP-00512-0007-2006.

the size of the neighborhood that is affected by the failure of a node. For example, if node p_i fails, we would prefer that this will not affect the ability of nodes far away from p_i to make progress. Failure locality allows us to quantify how "distributed" the decision-making is.

The other criterion we use is the *response time*, which is the time delay between when a node requests to enter its critical section and when it subsequently enters its critical section. In order to define response time, we assume upper bounds on the time that any node spends in the critical section and on the delay of any message.

To the best of our knowledge, there is no previous work on the local mutual exclusion for MANETs. The local mutual exclusion problem in static, wired distributed systems is known as the *dining philosophers (DP)* problem [5] and has been extensively studied [12]. Choy and Singh [3, 4] showed that 2 is the tight bound on failure locality for DP problem. The best previously known response time with optimal failure locality is $O(n^2)$ [13], where n is the number of nodes in the system. For any failure locality, the best previously known response time is $O(\delta^2)$ [4], where δ is the maximum degree of any node in the network (this algorithm has failure locality 4).

These previous algorithms are quite sophisticated, due to the difficulty of obtaining efficient solutions for the problem, even in the static case. Our contribution is to consider the problem in an even harsher environment, where nodes can move. We are able to obtain two efficient algorithms that exhibit different tradeoffs between failure locality and response time.

In more detail, our first result is a modular local mutual exclusion algorithm for MANETs, which assigns colors to the nodes and resolves conflicts based on the colors, as in the algorithms in [4]. We use the technique of "doorways", originally introduced by Lamport [9] and elaborated on by [4], to ensure local progress. We overcome the difficulties arising from node mobility by carefully choosing the sets of nodes that interact with each other based on their status with respect to motion. Because of node mobility, at certain times, nodes choose new colors for themselves.

We present two variations for the coloring module, demonstrating a trade-off between more practical implementation and better complexity properties. One version of the first algorithm is based on a simple greedy coloring; its failure locality is n and response time is $O((n+\delta^3)\delta)$. The second version is based on Linial's result [10], and relies on nodes knowing n and δ; it has much better failure locality $\max(\log^* n, 4) + 2$ and response time $O((\log^* n + \delta^4)\delta)$. In this version, although the failure locality is not optimal, it starts at 6 and grows very slowly with system size; the response time depends only weakly on n and is polynomial in δ, which makes this algorithm fast in sparse networks.

Our second algorithm achieves the optimal failure locality 2 and has response time $O(n^2)$; this matches the best known result for the static case [13], while tolerating node movement. Another important feature of this algorithm is that in the static case, the response time is $O(n)$, thus outperforming the best previously known result. Instead of resolving conflicts using colors, this algorithm uses a link reversal technique (first proposed by [7] and used in the context of dining philosophers by [2]). Our use of the preemptive fork-collection strategy of [13] allows us to reduce the failure-locality relative to [2]. Additionally, this algorithm does not require nodes to know the value of n and δ.

Table 1 compares our algorithms with previous results.

Due to space limitation, many implementation and proof correctness details are omitted. See [8] for the full details.

2 Preliminaries

The system consists of a set of nodes, communicating by exchanging messages over a wireless network. The communication network is modelled as a dynamically changing, not necessary connected, undirected graph, where mobile nodes of the system are represented as nodes in the graph, and there is an edge between two nodes that can communicate directly. Each node has a unique identifier, *ID*, and executes asynchronously at some unknown speed. Only nodes close to each other, called *neighbors*, may communicate directly. The nodes may fail by crash; a node does not change its location after a failure.

Communication links are bidirectional, reliable and FIFO. A link-level protocol ensures that each node is aware of the set of its neighbors by providing indications of link formations and failures. The same assumptions have been made by previous works on MANETs, e.g., [14]. In addition, we assume that a link between two nodes may be created or broken only when at least one of them moves, i.e., links do not fail or appear between static nodes.

An important assumption we make is that, when a link is created between two nodes, some mechanism breaks symmetry in a way that is biased toward nodes that are not moving. In more detail, we assume the link-level protocol supports two types of link creation notification, one for static nodes and another for moving nodes. If a link comes up between a static node and a moving node, the notifications are as expected. If a link comes up between two moving nodes, then exactly one of them gets the notification for a static node, e.g., according to their ID's. Possible implementation of such mechanism may require moving nodes to send special *start* and *stop* messages when they start and stop to move, respectively, and *heart-beat* messages during their movement, as presented by Wu and Li [15].

We assume an upper bound τ on the time that a node spends in its critical section. The total time for preparing, transmitting and receiving a message is assumed to be upper

Algorithm	failure locality	mobile response time	static response time	requirements
Choy and Singh [4]	4	N/A	$O(\delta^2)$	initial valid coloring
Tsay and Bagrodia [13]	2	N/A	$O(n^2)$	
Algorithm 1a	n	$O((n+\delta^3)\delta)$	$O((n+\delta^2)\delta)$	
Algorithm 1b	$\max(\log^* n, 4)+2$	$O((\log^* n+\delta^4)\delta)$	$O((\log^* n+\delta^3)\delta)$	know n and δ
Algorithm 2	2	$O(n^2)$	$O(n)$	

Table 1. Comparison of algorithms. Static and mobile response times refer to response times achieved when the algorithms are run in static and mobile setting, respectively.

bounded by ν. These bounds are unknown to nodes and used only for the analysis of our algorithms.

Each node in the system may be in one of three states: *thinking*, *hungry* and *eating*. These states correspond to the *remainder*, *trying* and *critical* code sections respectively in the classical notion of the mutual exclusion problem. Every node cycles between these three states. Some application external to our algorithm can change the state to *hungry* at any time if the current state is *thinking*, or to *thinking* if the current state is *eating*; our algorithms, under the right circumstances, change the state of a node to *eating* if it is *hungry*. In addition, in order to preserve the safety condition defined below, our algorithms may change the state of an *eating* node back to *hungry*; this may happen when a new link is created between two nodes. The transition from *eating* to *thinking* contains an *exit* code executed by a node upon its exit from the critical section.

The set of neighbors of each node p_i is stored in a local variable that is updated by a lower level protocol whenever the neighborhood of the node changes. The maximum degree of a node is denoted δ, while the total number of nodes is denoted n.

The *distance* of node p_i from node p_j is the number of links in the shortest path between p_i and p_j in the communication graph. The m*-neighborhood* of a node p_i includes all nodes in distance m or less from p_i.

A solution to the *local mutual exclusion problem* has to ensure (local) *mutual exclusion*: at any state of the system, no two neighbors are in the critical section.

We formally define the two measures used to evaluate the modules of our algorithms.

Definition 1. *A module has* failure locality *of* m *and* response time *of* T *if any node* p_i *that starts executing the module and remains static for* T *time units, finishes its execution within* T *time units, as long as there are no failures in the* m*-neighborhood of* p_i.

Note that if an algorithm has a finite response time, then it also guarantees *starvation freedom* for static nodes sufficiently far from failed nodes.

The definition is adapted to local mutual exclusion algorithms, which do not terminate, by considering the time interval between a node becoming hungry and starting to eat.

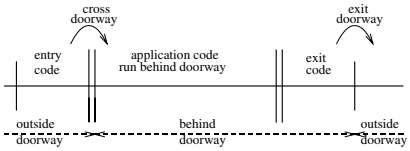

Figure 1. Schematic view of a doorway.

3 Doorways

Our first algorithm uses the concept of *doorways*, introduced by Lamport [9]. A doorway includes two code fragments, *entry* and *exit*. A node *crosses* the doorway when it completes the execution of the entry code, and *exits* the doorway when it completes the execution of the exit code (Figure 1). A node is said to be *behind* the doorway if it crossed the doorway, but has not exited yet; otherwise, it is *outside* the doorway.

The doorway guarantees that if a node p_i crosses it before its neighbor p_j begins executing the entry code, then p_j does not cross the doorway until p_i exits the doorway.

There are two kinds of doorways, called *synchronous* and *asynchronous*, differing in the way a node wishing to cross the doorway checks the state of its neighbors. In a *synchronous* doorway, a node crosses the doorway when all of its neighbors are observed to be outside the doorway *simultaneously*. In an *asynchronous* doorway, a node crosses the doorway once it finds each of its neighbors outside the doorway at least once, that is, the state of each neighbor is checked *independently*.

A *double doorway* [4] is a synchronous doorway within an asynchronous one. The entry code of this doorway consists of the entry code of an asynchronous doorway followed sequentially by the entry code of a synchronous one. In the exit code, the order of composition is reversed—the exit code of the synchronous doorway is followed by the exit code of the asynchronous one. The double doorway prevents the starvation possible in the entry of a syn-

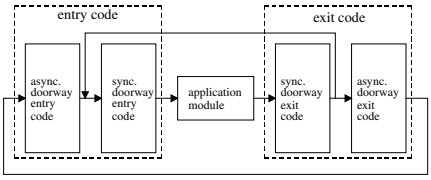

Figure 2. Double doorway with a return path. When a node exits the internal synchronous doorway, it may decide to exit the double doorway or to return to the entry code of the internal synchronous doorway.

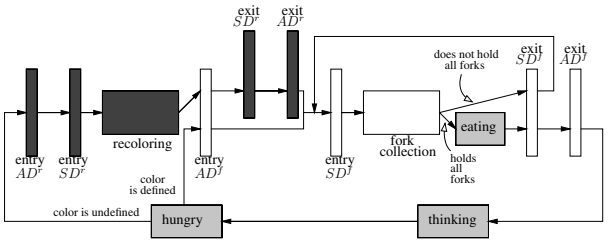

Figure 3. The structure of the first algorithm.

chronous doorway, yet ensures that after a period of time no new neighbors cross the doorway and execute concurrently with a node behind the doorway. This property is important for our first local mutual exclusion algorithm.

The following lemma states the key properties of a double doorway.

Lemma 1. *A node that starts executing the entry code of a double doorway at time t and remains static will exit the doorway by time $t + O(\delta T)$ if no node in its $(m + 2)$-neighborhood fails, provided the module executed inside the doorway has time complexity T and failure locality m.*

Our algorithm also uses a modification of a double doorway, called a *double doorway with a return path*, in which a node may return to the entry code of the synchronous doorway immediately after completing the exit code of this doorway (Figure 2). The decision is taken by the application module running behind the double doorway. Clearly, the number of returns has to be limited to prevent starvation of nodes waiting at the entry of both the asynchronous and synchronous doorways. Note that if the return path is never taken, this doorway is identical to the double doorway.

The following lemma states the key properties of a double doorway with a return path.

Lemma 2. *A node that starts executing the entry code of a double doorway with a return path at time t and remains static will exit the doorway by time $t + O(\delta T R)$ if no node in its $(m + 2)$-neighborhood fails, provided the module executed inside the doorway has time complexity T, failure locality m, and a node may execute the entry code of the synchronous doorway at most R times until it exits the double doorway.*

4 An algorithm with $O(\log^* n)$ failure locality

Our first algorithm uses *forks* as a metaphor for a shared resource between two neighbors, in our case, the use of the critical section. The fork is either owned by one of the neighboring nodes, or is in transit from one node to another. A node owning a fork can be viewed as having permission from its neighbor to enter the critical section. Initial distribution of forks must ensure that no two neighbors hold the same fork. Forks are destroyed when a link between two neighbors fails; when a link is created between two nodes, a corresponding fork, owned by the static node, is created. We assume that this operation is executed by a link-level protocol and omit further description of this procedure.

A node that becomes hungry requests missing forks from its neighbors. The algorithm uses *colors* from an ordered set represented by integers in order to resolve conflicts between neighboring nodes that are hungry simultaneously by setting a node with a smaller color to have a higher priority. Nodes competing for access to the critical section must be legally colored, i.e., no two neighbors have the same color. In a mobile setting, however, even if legal colors are initially pre-computed for the nodes, nodes may move from one location to another and violate the legality of coloring. As a result, we need to recolor moving nodes before they start competing for access to the critical section.

The algorithm sequentially executes a *recoloring* module and a *fork collection* module, each within a double doorway, one of them having a return path (Figure 3).

The recoloring module is executed by a hungry node that has moved to a new neighborhood; it guarantees that the node obtains a new legal color in a relatively small range. A simple way to guarantee the legal coloring is to color each node with its ID; unfortunately, the resulting number of colors does not reflect the current topology of the network. We guarantee that the number of colors is proportional to δ, the maximum degree of any node in the system, by running, behind a double doorway, an algorithm which ends by picking a new color for a node from a small range. We provide two different implementations for this algorithm.

The recoloring module is also executed by each node to obtain an initial color; we do not discuss this issue further.

The fork collection module is executed when a node crosses the second double doorway. A hungry node may arrive at the entrance of this doorway with or without executing the recoloring module, depending on whether or not it moved after last eating. A node that crosses the asyn-

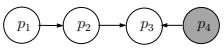

Figure 4. Edges are directed from nodes with larger colors to nodes with smaller colors.

chronous doorway AD^f without executing the recoloring module, does not execute the exit code of the first double doorway and continues directly to the entry code of SD^f. A node executing the fork collection module gathers forks required to access the critical section. The correctness of this module relies on the legality of the coloring of nodes crossing the second double doorway, while its performance depends on the range of colors.

When a moving node arrives at a new neighborhood, it exits any doorway that it has crossed and waits for messages from its new static neighbors with their color and logical position regarding the doorways. A static node with a new neighbor sends a message with its color and position relative to the doorways and continues its execution. After receiving messages with positions of all new (static) neighbors, the newly arrived node executes the recoloring module, when it next becomes hungry. The doorways in both modules do not let newly arrived nodes interfere with static nodes entering the critical section, keeping them blocked at the entrance of the first or the second double doorway, depending on the status of their hungry static neighbors.

4.1 The fork collection module

A node p_i enters the fork collection module after crossing the second double doorway. It first sends requests for missing forks to its *low* neighbors—those with a smaller color—and then to *high* ones—those with a larger color.

In more detail, p_i first sends requests for all its missing *low* forks (forks shared with its low neighbors). While waiting for them, p_i grants any request for a fork, although when it relinquishes a low fork, it indicates that it wants it back. Once p_i acquires all its low forks, it sends requests for all its missing *high* forks (forks shared with its high neighbors). While waiting for them, p_i postpones granting any requests for high forks, unless it gets a request for a low fork, in which case it grants that request as well as all suspended requests for high forks. Having collected all forks, a node enters the critical section. When it exits the critical section, a node responds to all delayed requests with a fork and exits the double doorway.

When a node p_j moves causing the link between p_i and p_j to fail, p_i is able to proceed with fork collection, even in situations where it was blocked before p_j moved. For example, Figure 4 shows a part of a system consisting of four nodes, one of which, p_4, crashes. If p_3 gathers all forks from its low neighbors, but does not get a fork from its failed high neighbor p_4, it will suspend any request from p_2. Thus p_2 responds to any request from p_1 with a fork and will never ask for it back, thus protecting other nodes from p_4's failure, at the price of being blocked. In our setting, however, p_3 may move, in which case p_2 can proceed with fork collection by asking for the fork back from p_1. This scenario may increase the response time of the fork collection module and is handled by the return path of the doorway: a node that detects a link failure to a low neighbor, which holds their shared fork, exits the synchronous doorway, releasing all requested forks, and returns to the entry code of this doorway. This transition is represented in Figure 3 by a path from the exit of SD^f back to the entry of SD^f. In the example, when p_2 detects that p_3 has moved, it exits the synchronous doorway, without asking for a fork from p_1; p_1 proceeds with fork collection as if p_3 has not moved away.

4.2 Pseudo-code

The first local mutual exclusion algorithm is presented in Algorithm 1. Each node p_i has the following local variables:

- N, set of IDs of neighboring nodes. Maintained by the lower level according to notifications about link creations and failures.
- $color[j]$, array of colors of a node and its neighbors; initially undefined. Updated when *update-color* message with a new color is received or when p_i picks a new color for itself.
- $state$, takes on values *hungry*, *eating*, or *thinking*; initially *thinking*.
- S, set of neighbors with suspended forks requests; initially empty.
- $at[j]$, boolean flag, indicating that p_i holds the fork shared with p_j; initially *true* when $ID[i] < ID[j]$.
- R, set of neighbors behind the first double doorway that are participating in the recoloring; initially empty.
- $L[j]$, stores the last doorway-related (cross/exit) message that p_i received from p_j; initially $exit_{all}$.

The code uses the macros *all-(low)-forks* to indicate that the node has all its (low) shared forks.

A node enters the fork collection module when it crosses the second double doorway. The actions taken by a node executing this module are presented in Lines 1–35; an overview was given in Section 4.1.

Upon exit from the critical section (Line 5), a node recolors itself with the smallest non-negative color not used by any of its neighbors, as indicated in $color[]$ array; this color is legal since it is chosen in exclusion. The color is clearly in $[0..\delta]$, which improves future executions of the fork collection module in cases when the recoloring module picks

Algorithm 1 The first LME algorithm, code for p_i

Fork collection module

1: when SD^f is crossed (and $state$ is *hungry*) :
2: **if** all-forks **then** $state := eating$
3: **if** all-low-forks **then** `request-high-forks()`
4: **else** `request-low-forks()`

5: when $state$ is set to *thinking*:
6: $color[i]$:= the smallest non-negative color not used by any neighbor
7: broadcast *update-color*$(color[i])$ msg
8: **for each** j s.t. $j \in S$ **do**: `send-fork(`j`)`
9: $\langle$ exit SD^f and AD^f $\rangle$

10: when req msg is received from j and $at[j]$:
11: **if** $[(color[j] > color[i]) \wedge$
12: $(\neg$all-low-forks $\vee$ *outside* $SD^f)]$ **then** `send-fork(`j`)`
13: **else if** $[(color[j] < color[i]) \wedge$
14: $(\neg$all-forks $\vee$ *outside* $SD^f)]$ **then**
15: `send-fork(`j`); release-high-forks()`
16: **else** $S := S \cup \{j\}$

17: when $\langle fork, flag \rangle$ msg is received from j:
18: $at[j] :=$ *true*
19: **if** all-forks **then** $state := eating$
20: **if** all-low-forks **then**
21: **if** $flag$ **then** $S := S \cup \{j\}$
22: `request-high-forks()`
23: **else if** $flag$ **then** `send-fork(`j`)`

24: `request-low-forks()`:
25: **for each** $j \in N$ s.t. $(color[j] < color[i]) \wedge \neg at[j]$ **do**:
26: send req msg to j

27: `request-high-forks()`:
28: **for each** $j \in N$ s.t. $(color[j] > color[i]) \wedge \neg at[j]$ **do**:
29: send req msg to j

30: `send-fork(`j`)`:
31: send $\langle fork, ((color[j] < color[i]) \wedge$ *behind* $SD^f)\rangle$ to j
32: $at[j] :=$ *false*; $S := S \setminus \{j\}$

33: `release-high-forks()`:
34: **for each** $j \in S$ s.t. $(color[j] > color[i]) \wedge at[j]$ **do**:
35: `send-fork(`j`)`

Wrapper code for the recoloring module

36: when SD^r is crossed (and $state$ is *hungry*):
37: $R := N$
38: $color[i] := (-1) * $ `new-color()` $- 1$
39: broadcast *update-color*$(color[i])$ msg

40: when some msg sent from `new-color()` is received from $j \wedge$ *outside* SD^r:
41: send NACK to j

42: when NACK msg is received from j:
43: $R := R \setminus \{j\}$

Link creation and failure handling

44: $LinkUp(j)$ indication arrives while static:
45: $at[j] :=$ *true*; $color[j] :=\perp$; $L[j] := exit_{all}$
46: send $\langle$*update-color*$(color[i]), L[i]\rangle$ msg to j

47: $LinkUp(j)$ indication arrives while moving:
48: $at[j] :=$ *false*; $color[j] :=\perp$
49: **if** $\langle$ behind SD^f $\rangle$ **then**
50: **if** $(state = eating)$ **then** $state := hungry$
51: **for each** j s.t. $j \in S$ **do**: `send-fork(`j`)`
52: $\langle$ exit any doorway $\rangle$
53: **wait until** $\langle$*update-color*$(color[j]), L[j]\rangle$ msg is received
54: **if** $(state = hungry)$ **then**
55: $\langle$ go to the entry of $AD^r\rangle$

56: $LinkDown(j)$ indication arrives:
57: **if** $\langle$ behind SD^f $\rangle$ **then**
58: $S := S \setminus \{j\}$
59: **if** $(\neg at[j] \wedge (color[j] < color[i]))$ **then**
60: $\langle$ exit SD^f doorway and return to its entry $\rangle$
61: **else if** $\langle$ behind SD^r $\rangle$ **then** $R := R \setminus \{j\}$

colors in a range of a greater size. The new color is sent to all neighbors (Line 7).

A node enters the recoloring module when it crosses the first double doorway. The recoloring module consists of the wrapper code presented in Lines 36–43, and the coloring procedure `new-color()` that calculates new colors for nodes, presented in two variations in Section 4.4.

A node sets its new color to one less than the negative of the value returned from `new-color()` (Line 38). This operation guarantees that colors chosen by the recoloring module are negative, leaving the colors $0, 1, \ldots, \delta$ for nodes exiting the critical section (recall Line 6). The code in Lines 40–43 handles recoloring-related messages received by or from nodes that are not participating in recoloring.

An indication of link creation is treated differently by static nodes (Lines 44–46) and moving nodes (Lines 47–55). When a static node p_i receives an indication that a link to p_j is created, it marks itself as the owner of the shared fork and initializes the color of the moving node to $\perp$; this value will change once the new neighbor chooses a color and notifies p_i. In addition, p_i sets $L[j]$ to $exit_{all}$, meaning it considers p_j to be outside of any doorway. Finally, p_i sends its color and doorway status to p_j.

A node p_i that has just arrived in the neighborhood of a static node p_j, sets $at[j]$ to *false* and its color to $\perp$ (Line 48). If p_i was behind SD^f (running the fork collection module), it changes its state to hungry if it was eating (Line 50) and releases all forks with suspended requests (Line 51). Next, it notifies all its neighbors that it has exited any doorway it may have crossed; this ensures all neighbors have a consistent view of p_i's location with respect to the doorways. Finally, p_i waits for the messages from its new neighbors (sent in Line 46), and if it is hungry it starts recoloring.

Lines 56–61 handle link failure due to node movement. If the node is behind the SD^f doorway and the notification comes from a low neighbor holding the shared fork (cf. the scenario of Figure 4), then it exits the synchronous doorway and returns to its entry code (Lines 59–60).

4.3 Correctness proof

Due to space limitation, most of this section is omitted. See [8] for the full details.

A node accesses the critical section in the fork collection module and it must have all forks before that. Since when a link is created between two nodes, a single fork is created which is held by exactly one node and the fork is destroyed upon destruction of the link, it can be shown that the first algorithm satisfies the local mutual exclusion property.

We formulate an assumption on the properties of the recoloring module, which allows us to prove that the colors of nodes running the fork collection module are legal. This serves as an interface between the two modules.

Assumption 1. *Any neighbor of p_i that starts the recoloring module when p_i is behind the first double doorway, picks a different color than p_i's.*

The following lemma exploits the interleaving of the two double doorways.

Lemma 3. *If p_i is behind the second double doorway in the time interval $[t_1, t_2]$, then for any neighbor p_j that is behind the same doorway at time $t \in [t_1, t_2]$, $color[i] \neq color[j]$.*

Suppose p_i crosses SD^f at time t and let Δ be the maximal absolute value of a color produced by the recoloring module. In the full paper, we prove that if p_i remains behind SD^f throughout the interval $[t, t+X]$, and no nodes within a distance of 2 from p_i fail, then p_i starts eating within X time units, where X is $O(\Delta + \delta)$. For the proof, we define a graph LG_i, consisting of p_i and all hungry nodes that are also behind SD^f, have higher priority than p_i and have a path in the network to p_i. The use of the doorway ensures that after a certain period of time depending on the color of p_i, nodes cease joining LG_i. Thus the number of nodes with higher priority, which may delay p_i while running the fork collection module simultaneously, is limited.

Denote the time complexity of the recoloring module, that is the number of time units from the time a node crosses the first double doorway till it finds a new color, by T_c and its failure locality by f_c. The following lemma summarizes the properties of the first algorithm:

Lemma 4. *The first algorithm has failure locality* $\max(f_c, 4) + 2$ *and response time* $O((T_c + \delta^2(\Delta + \delta))\delta)$.

In the static case, since the return path of the second double doorway is never taken, the response time of the first algorithm is $O((T_c + \delta(\Delta + \delta))\delta)$. Moreover, after all nodes in the static system set a color upon the exit from the critical section (in Line 6), the recoloring module is never run again; the response time in this special case becomes $O(\delta^2)$.

4.4 Two coloring procedures

We present two versions of the coloring procedure `new-color()` used by the wrapper code in Algorithm 1, demonstrating a trade-off between a more practical implementation and better (theoretical) complexity properties. As can be seen from Lemma 4, these properties are affected by the failure locality, running time and the color range of the coloring procedure.

The first coloring procedure is a simple greedy algorithm, in which a node iteratively exchanges messages with its neighbors to obtain a view of the subgraph that includes all nodes performing recoloring concurrently; the node then runs a local predefined coloring algorithm on that subgraph (e.g., DFS starting from a node with smallest ID) and picks a new color in a greedy manner, depending on its neighbors.

By proving that two neighbors running recoloring simultaneously obtain the same subgraph, we are able to show that this procedure satisfies Assumption 1, produces colors in a range of $O(\delta)$, and has failure locality n and response time $O(n)$. Substituting $T_c = O(n)$, $f_c = n$ and $\Delta = O(\delta)$ in Lemma 4, we get the next theorem:

Theorem 5. *When run with the greedy coloring procedure, the first algorithm satisfies the local mutual exclusion property in a MANET, has failure locality n and response time $O((n + \delta^3)\delta)$.*

The alternative coloring procedure is more sophisticated, implementing an adaptation of the fast graph coloring scheme of Linial [10]. This procedure assumes knowledge of n and δ and works in iterations as well; in each iteration, a node learns the colors of its neighbors that are behind the double doorway of the recoloring module, and calculates a new color for itself from a smaller range. The smaller range of colors suffices due to the following combinatorial result:

Theorem 6 (Erdõs, Frankl and Füredi [6])**.** *For any two integers n and δ with $n > \delta$, there is a set J of n subsets of $\{1, \cdots, \lceil 5\delta^2 \log n \rceil\}$ s.t. for any $\delta + 1$ subsets F_0, F_1, ..., $F_\delta \in J$, $F_0 \nsubseteq \bigcup_1^\delta F_i$.*

This result is applied in each iteration by considering subsets of colors and having each node choose the color that does not appear in any of the subsets of colors corresponding to neighbors running the recoloring module. We prove that the final chosen colors are legal (i.e., the procedure satisfies Assumption 1) and in the range of size $O(\delta^2)$, while the failure locality is $\log^* n$ and the response time is $O(\log^* n)$. Substituting $T_c = O(\log^* n)$, $f_c = \log^* n$ and $\Delta = O(\delta^2)$ in Lemma 4, we get the next result:

Theorem 7. *When run with the alternative coloring procedure, the first algorithm satisfies the local mutual exclusion property in a MANET, has failure locality* $\max(\log^* n, 4) + 2$ *and response time* $O((\log^* n + \delta^4)\delta)$.

Thus, the alternative coloring achieves much better properties at the price of increased combinatorial complexity, heavy pre-computation and the assumption of knowing n and δ. Since recoloring is only required when a node moves (and for initialization), the simple greedy coloring procedure, although not achieving as good theoretical complexity bounds, is a practical choice, especially for systems in which nodes do not move or fail very frequently.

5 An algorithm with optimal failure locality

Our second local mutual exclusion algorithm for MANETs has optimal failure locality 2, but its response time depends quadratically on n. Besides having better failure locality, this algorithm does not require nodes to know

the values of n or δ as the second version of the first algorithm does. This makes it more flexible when new nodes join the system or there are unpredictable topology changes, preventing δ from being known in advance. In addition, when it is run in a static environment, it achieves response time linear in n, which is better than any other known solution with optimal failure locality.

The solution uses a modified fork collection mechanism without doorways or colors. The idea is to use dynamic priorities which change when nodes enter their critical sections [13]. The role of colors is replaced by an array of flags, denoted $higher$, identifying if a neighbor has a higher priority over the node. A node that exits the critical section resets the values in its $higher$ array to $true$, thus lowering its priority relative to its neighbors.

When a node becomes hungry, it sends a *notification* message to its neighbors. A node p_j that receives a notification from p_i, lowers its priority only if it is thinking and it has higher priority than p_i. The motivation behind this notification mechanism is to prevent a node p_j that has higher priority and is currently thinking from interfering with the fork collection of p_i by becoming hungry later.

The following theorems summarizes the properties of the second algorithm in mobile and static setting, respectively.

Theorem 8. *The second algorithm satisfies local mutual exclusion property in a MANET, has failure locality* 2 *and response time* $O(n^2)$.

Theorem 9. *When there are no changes in the topology, the second algorithm has response time* $O(n)$.

This improves on the best previously known algorithm for the dining philosophers problem in static systems with optimal failure locality [13], which achieved $O(n^2)$ response time. The improvement is achieved due to the notification mechanism, described above.

6 Discussion

We have proposed the local mutual exclusion problem, an extension of the dining philosophers problem from static networks, as a potentially useful primitive for building applications for mobile ad hoc networks. Two algorithms that exhibit different tradeoffs between failure locality and response time were presented.

It would be interesting to investigate the inherent complexity of local mutual exclusion. The result of Linial [10] implies a lower bound of $\Omega(\log^* n)$ on response time for the dining philosophers problem in static networks when there is no initial coloring of the nodes; this lower bound clearly carries over to the mobile case. Thus, one of the versions of our first algorithm has a response time close to optimal. Choy and Singh [4] obtain response time that does not depend on n, but assume a predefined coloring of the nodes; our recoloring module can be employed to efficiently and distributively pre-compute this coloring in a static setting.

Additional future work could be to explore other performance measures, such as message complexity, and to find more applications of local mutual exclusion.

References

[1] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proc. of IEEE ISCC*, pages 539–545, 2002.

[2] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[3] M. Choy and A. K. Singh. Tight lower bounds on failure locality of distributed synchronization. In *Proc. of Allerton Conf. Comm., Control, and Comp.*, pages 127–136, 1992.

[4] M. Choy and A. K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Trans. Program. Lang. Syst.*, 17(3):535–559, 1995.

[5] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] P. Erdõs, P. Frankl, and Z. Füredi. Families of finite sets in which no set is covered by the union of r others. *Israel J. Math*, 51:79–89, 1985.

[7] E. Gafni and D. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Trans. on Comm.*, 29(1):11–18, 1981.

[8] A. Kogan. Efficient and robust local mutual exclusion in mobile ad hoc networks. Master's thesis, Department of Computer Science, Technion, 2008.

[9] L. Lamport. On interprocess communication (part I and II). *Distributed Computing*, 1, 2(2):77–101, 1986.

[10] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[11] R. Mellier and J.-F. Myoupo. Fault tolerant mutual and k-mutual exclusion algorithms for single-hop mobile ad hoc networks. *Int. J. of Ad Hoc and Ubiquitous Computing*, 1(3):156–166, 2006.

[12] M. Papatriantafilou and P. Tsigas. On distributed resource handling: Dining, drinking and mobile philosophers. In *Proc. of OPODIS*, pages 293–308, 1997.

[13] Y.-K. Tsay and R. Bagrodia. An algorithm with optimal failure locality for the dining philosophers problem. In *Proc. of WDAG*, pages 296–310, 1994.

[14] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7(6):585–600, 2001.

[15] J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proc. of DIALM*, pages 7–14, 1999.

[16] W. Wu, J. Cao, and M. Raynal. A dual-token-based fault tolerant mutual exclusion algorithm for manets. In *Proc. of MSN*, pages 572–583, 2007.