Efficient and Robust Local Mutual Exclusion in Mobile Ad Hoc Networks

Alex Kogan

Efficient and Robust Local Mutual Exclusion in Mobile Ad Hoc Networks

Research Thesis

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in Computer Science

Alex Kogan

Submitted to the Senate of the Technion – Israel Institute of Technology

EYAR, 5768 HAIFA June, 2008

The research thesis was done under the supervision of Prof. Hagit Attiya in the Department of Computer Science.

I would like to express my greatest thanks to Hagit for her incredible guidance and for giving me freedom to explore, without letting me get lost. I also show her my gratitude for making the research so interesting, challenging and exciting, and for her support, both academical and financial, during my Master's degree studies.

I would like to sincerely thank Prof. Jennifer Welch for her invaluable contribution to this research, for dedicating her precious time far beyond I could ever expect, for warm hospitality at Texas A&M University and for always being available for my endless number of questions, comments and reviews.

I would also like to thank my dear family. I thank my beloved wife Ella for standing by me during all my graduate studies, encouraging me during the downs and sharing my happiness during the ups. I thank my parents who always believed in me and helped in whichever way they could to push me as far as I could only imagine; I owe all my achievements to them. I also thank my grandmother, my sister and my parents-in-law for extending their support and love in every possible way.

Last, but not least, I want to thank Prof. Roy Friedman, Prof. Shlomi Dolev and Prof. Reuven Cohen for helpful and valuable comments.

The generous financial help of the Technion is gratefully acknowledged.

Contents

Abstract **1**

1 Introduction **3**

2 Related work **7**

3 Preliminaries **11**

3.1 System model 11

3.2 Problem definition 12

4 Doorways **15**

5 An algorithm with $O(\log^* n)$ failure locality **19**

5.1 The fork collection module 20

5.2 Pseudo-code 21

5.3 Correctness proof 24

5.4 The coloring procedures 31

5.4.1 Greedy coloring 32

5.4.2 More efficient coloring 34

6 An algorithm with optimal failure locality **39**

6.1 Pseudo-code 39

6.2 Correctness proof 40

7 Discussion **45**

List of Figures

1 Schematic view of a doorway. 15
2 Basic doorway implementations. 16
3 Double doorway. 17
4 Double doorway with a return path. 18

5 Schematic description of the first algorithm. 20
6 Example for a scenario that may increase the response time in the fork collection module due to nodes mobility. 21

List of Tables

1 Comparison of algorithms. 5

Abstract

In a mobile ad hoc network, nodes that are geographically close may need to compete for exclusive access to a shared resource. This thesis studies an abstraction of this problem, called *local mutual exclusion*, an extension of the dining philosophers problem, well-studied in static networks, to mobile networks. A desirable feature of an algorithm for this problem is to limit the negative effect of a crashed node to a small neighborhood around the crash; the size of this neighborhood is called the *failure locality*. Another requirement from such algorithm is having the *response time* independent of the total number of nodes, thus providing a scalable solution.

The thesis presents two algorithms, exhibiting different tradeoffs between simplicity, failure locality and response time. 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 other variation has worse performance, but simpler implementation. The second algorithm has optimal failure locality and response time that is quadratic in the number of nodes. A pleasing aspect of the latter algorithm is that when nodes do not move, it has linear response time, improving on previous results for static algorithms with optimal failure locality.

Chapter 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 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., [3,7,26,39,41,42]), it appears to have fewer potential applications than the problem studied in this thesis.

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 [26]) or the control over 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 [9], which measures 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 [10] and has been extensively studied (see [29]). Choy and Singh [8, 9] showed that 2 is the tight bound on failure locality for DP problem. The best previously known response time with optimal failure locality is $O(n^2)$ [35, 37], where n is the number of nodes in the system. For any failure locality, the best previously known response time is $O(\delta^2)$ [9], where δ is the maximum degree of any node in the network (this algorithm has failure locality 4).

All the 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 [9]. We use the technique of "doorways", originally introduced by Lamport [20] and elaborated on by [9], 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 versions of the coloring module, one using a simple greedy coloring algorithm, and the other using a fast coloring algorithm of Linial [22]. The two coloring modules demonstrate a trade-off between more practical implementation and better complexity properties. The version of the first algorithm based on the greedy coloring has poor failure locality $O(n)$ and response time $O((n+\delta^3)\delta)$. The second version based on Linial's result [22] relies on nodes having knowledge of n and δ and has much better failure locality $\max(\log^* n, 4) + 2$ and response time $O((\log^* n + \delta^4)\delta)$. In the second 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; this matches the best result for the static case, while tolerating node movement. Its response time is $O(n^2)$, matching the best known result for the static case. 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 Gafni and Bertsekas [15] and

algorithm	failure locality	mobile response time	static response time	requirements
Choy and Singh [9]	4	N/A	$O(\delta^2)$	initial valid coloring
Tsay and Bagrodia [37]	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.

used in the context of dining philosophers by Chandy and Misra [6] to obtain an algorithm with failure locality n.). Our use of the preemptive fork-collection strategy of Choy and Singh [9] allows us to reduce the failure-locality relative to Chandy and Misra [6]. Additionally, this algorithm does not require nodes to know the value of n and δ.

Table 1 compares our algorithms with others.

The rest of this thesis is organized as following: the related work is described in Chapter 2. The system model and problem definition are given in Chapter 3. Chapter 4 presents doorways, which are used by the first local mutual exclusion algorithm presented in Chapter 5. Chapter 6 presents the second local mutual exclusion algorithm. We conclude and discuss further research directions in Chapter 7.

Chapter 2

Related work

The dining philosophers problem was extensively explored in static environments. Most of the past work is summarized in [29]. Choy and Singh [9] define the notion of failure locality and show that it must be at least 2. They also present two algorithms with constant failure locality, which use doorways [20] to ensure local progress of nodes. One of these algorithms achieves failure locality 3 and response time exponential in δ, while the other has response time polynomial in δ at the cost of worse failure locality 4. In another paper [8], Choy and Singh present an algorithm with optimal failure locality, but with unacceptably high response time $O(\delta^{2n}n!)$. Tsay and Bagrodia [37] propose an algorithm for solving the dining philosophers problem with failure locality 2, response time $O(n)$ when all nodes function properly, and $O(n^2)$ when failures may occur. Sivilotti et al. [35] obtained the same results independently, but later.

Earlier work by Styer and Peterson [36] presents an algorithm with response time of $O(\delta^{\log\delta+1})$ and failure locality of $O(\log\delta)$, by extending the edge-coloring algorithm of Lynch [23]. Awerbuch and Saks [2] provide an algorithm with response time of $O(\delta^2\log U)$, where U is the range of node IDs, and failure locality of $O(\delta)$. The improved response time is achieved by using a distributed queue to schedule jobs competing for shared resources. Each job is assigned a position in the queue, which decreases until the job reaches the head of the queue and gets executed. Although competing jobs may be assigned initially the same position in the queue, the scheduling algorithm prohibits them from reaching the head of the queue simultaneously.

Recent work strives to reduce failure locality by using failure detectors. Pike and Sivilotti [31] use eventually perfect failure detector $\Diamond P$ to achieve failure locality 1. They prove that $\Diamond P$ is the optimal failure detector in the sense that any algorithm using $\Diamond P$ has failure locality at least 1 and this is the weakest detector in the Chandra-Toueg hierarchy [5] that may achieve this failure locality. The algorithm is based on ideas of Chandy and Misra [6] and uses doorways as in the work of Choy and Singh [9]. Later, Pike et al. [32] using a local refinement of the eventually perfect failure detector $\Diamond P_1$ and ideas from [6], provide an algorithm with failure locality 0 under assumption of eventual weak exclusion, which allows violation of mutual exclusion finitely many times until some point after which there are no violations. Since neither failure detector can be implemented in a purely asynchronous system, the use of either one indicates a stronger environment than that studied in this thesis.

Nesterenko and Arora [28] extend the ideas of Chandy and Misra [6] to give an algorithm that tolerates malicious crashes, while still achieving failure locality 2. In a *malicious crash*, a failed node behaves in an arbitrary way for a finite time and then ceases all operation undetectably by other nodes. The solution is implemented in the shared-memory model and guarantees that any node which is far enough (i.e., at least at distance 2) from crashed nodes, gets into the critical section, but only eventually (i.e., there is no fixed bound on the response time).

Since the dining philosophers problem cannot be solved in a symmetric deterministic fashion [21,23], i.e., in systems with indistinguishable nodes and resources, several randomized symmetric algorithms were proposed. The first work in this area, by Lehman and Rabin [21], considers the classical definition of the problem as a ring of philosophers, or nodes, in the shared-memory system and provides probabilistic lockout-free solutions. They assume nodes to crash only outside of their critical sections. Herescu and Palamidessi [17] generalize these ideas to provide a lockout-free solution for arbitrary topologies of nodes, based on the idea of letting nodes assign a random priority to their shared resources and thus establish a partial order on the resources, breaking the symmetry. Another extension, by Duflot et al. [13], removes the assumption that the scheduling of steps by nodes is fair (in a fair schedule, every node that continuously tries to access the critical section must be eventually scheduled to perform its next step). Their solution provides a weaker deadlock-freedom property.

Several papers propose randomized algorithms for the asymmetric version of the DP problem. Bar-Ilan and Peleg [4] present an algorithm for the synchronous static model that runs in phases and uses coin flips to determine which node will get access to the shared resources in each phase. The probability of a node p_i to get an access in some phase is $1/(\delta_i + 1)$, where δ_i is the degree of p_i in the network, and conflicts between nodes are resolved in favor of the one having higher degree. This solution has expected response time of $O(\delta)$, but the paper explicitly assumes that no failures occur in the system. Rhee et al. [33], using similar ideas, propose a randomized algorithm for semi-synchronous wireless sensor networks, in which nodes have clocks whose rates may drift. The expected response time and message complexity are $O(\delta)$. This work does not deal with node and/or link failures either.

Ha et al. [16] consider a simple randomized approach for the related problem of *multi-word locking* in the asynchronous shared memory system. They propose to acquire locks in a random order. This algorithm, though very simple to implement, has response time which depends poly-logarithmically on the number of registers (which would translate to poly-logarithmic dependence on the number of nodes in our model) and has only probabilistic guarantees.

The closest dynamic system model in which problems similar to DP were considered is that of Mayer et al. [25]. In their setting, nodes are static, but links may fail and come up between any pair of nodes. They consider a weaker version of the problem, denoted as (d, m)-DP, in which each node in a communication graph of minimum degree d has to acquire m resources for entering a critical section, where $m \leq \lceil d/2 \rceil$. The solution is constructed by a reduction to the general DP problem and applying Choy and Singh's approach [9]. This yields a very effective response time of $O(m)$, but has a non-trivial assumption that no link in a 5-neighborhood fails, which would translate in a MANET to the requirement that every node in the 5-neighborhood of a node trying to enter its critical section should not move.

Despite extensive research on upper bounds on the response time, little has been known about lower bounds. The obvious lower bound of $\Omega(\delta)$ is mentioned by several works, e.g., [2, 9, 37]. Rhee and Welch [34] slightly refine this bound to $\Omega(\chi)$, where χ is the chromatic number of the network. The authors also provide an algorithm that achieves their refined bound under the assumptions of an initial coloring of nodes and no failures occurring in the system.

Although we are not aware of prior work on local mutual exclusion in mobile ad-hoc networks, several works proposed solutions for related distributed problems in this setting. Walter et al. [39] solve mutual exclusion using a token and relying on Gafni and Bertsekas's partial reversal algorithm [15], which provides loop-free routing in a network subject to link failures. Only the token-holder may enter the critical section, while other nodes request access by sending messages to the token-holder. This approach prevents the algorithm from being tolerant to failures of nodes. In addition, the liveness property is ensured only when link failures cease throughout the system.

Walter et al. [38] present a solution to the problem of *k-mutual exclusion* in a MANET, where k nodes are allowed to be in the critical section simultaneously. Their approach extends the ideas of Walter et al. [39] by using k tokens. Yet another extension is presented by Jiang [18], who formulates and solves the problem of *h-out-of-k-mutual exclusion*, where each node may request up to h shared resources from k available in order to access the critical section.

Link reversal [15] was used in the context of several other problems in MANETs. Park and Corson [30] provide an efficient algorithm for routing in mobile ad-hoc networks. Malpani et al. [24], extending the ideas of Park and Corson, try to solve leader election problem in a group of nodes. They prove correctness of their algorithm for the case of a single topology change only.

Mellier et al. [27] present a non-token-based randomized algorithm for the mutual exclusion problem in MANETs. They construct a process which works in rounds and randomly splits the nodes trying to get into the critical section into two sets at each round, yielding a random binary tree. Each node participates in this process until it realizes that none of its neighbors appears in the same set, and then it enters the critical section. This solution assumes a synchronous single-hop network, and tolerates failures of nodes competing for the access to the critical section, but not failures of nodes executing the critical section. The authors extend their ideas to solve the k-mutual exclusion problem [26].

As stated before, our first algorithm implements recoloring module, which assigns colors to nodes in a way that no two neighboring nodes running the module receive the same color. Coloring nodes of a graph with small number of colors is one of the most fundamental problems in graph algorithms. The most seminal result is that of Linial [22], which shows a lower bound of $\Omega(\log^* n)$ communication rounds required to color a ring of n nodes with 3 colors in a synchronous static system. Linial shows an algorithm to color general graph with $O(\delta^2)$ colors in $O(\log^* n)$ rounds. This algorithm, adapted for asynchronous mobile setting, is used by one of the versions of our first algorithm.

Recently, Kuhn and Wattenhofer [19] slightly improved this lower bound and showed that $\Omega(\log^* n + \delta / \log^2 \delta)$ time is required in order to obtain an $O(\delta)$-coloring. They also explored the potential of randomization in coloring algorithms and showed an algorithm that colors any graph with $\delta + 1$ colors in $O(\delta \log \log n)$ time with probability $1 - 1/n^c$, where c is an arbitrary constant. Since $O(\delta^2)$-coloring can be deterministically converted to $(\delta + 1)$-coloring in $O(\delta \log \delta)$ rounds [19], their algorithm outperforms Linial's only when $\delta \gg \log n$.

Chapter 3

Preliminaries

3.1 System model

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 crashing; a node does not change its location after it fails.

An *execution* of the system is an infinite sequence of the form $C_0, \phi_1, C_1, \phi_2, ...$, where each C_k is a *configuration*, holding the state of each node in the system, and ϕ_i is an *event*, specifying one of the following types of events: (1) a local computation performed by some node, or (2) delivery of a message from some node to another, or (3) delivery of a notification about link creation or failure from a link-level protocol, occurring in some node. Furthermore, we associate a nonnegative real number with every event, specifying the real time at which the event occurs. The times start at 0 and are nondecreasing (for further details, refer to Section 2 in [1]).

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., [24, 38, 39]. In addition, we assume that a link between two nodes is 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 suggested by Wu and Li [40].

Although the processing speed of each node is unknown, 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 bounded by ν. These bounds are unknown to nodes and used only for the analysis of our algorithms.

3.2 Problem definition

Each node in the system may be in one of three sets of states: *thinking*, *hungry* and *eating*. These sets correspond to the *remainder*, *trying* and *critical* code sections respectively in the classical terminology of the mutual exclusion problem. A thinking state captures the situation when a node does not require access to its critical section; it is an initial state of each node in the system. A node enters a hungry state when it requests access to the critical section. Finally, an eating state represents the situation when a node has gotten access and entered its critical section. Every node cycles between these three sets of states. Some application external to our algorithm can change the state to *hungry* at any time if the current state is in *thinking* set, or to *thinking* if the current state is in *eating* set; our algorithms, under the right circumstances, change the state of a node to *eating* if it is in *hungry* set of states. 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 only when a node moves into new neighborhood and new links are created. 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. One version of our first algorithm assumes that both δ and n are known to all nodes in the system; the other version, as well as the second algorithm, does not require this assumption.

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 configuration of any execution 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.*

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

Chapter 4

Doorways

Our first algorithm uses the concept of *doorways*, introduced by Lamport [20]. 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 *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*. Figure 2 presents implementations of these two types of doorways.

The synchronous doorway ensures that once a node p_i crosses the doorway at time t, any neighbor that is outside the doorway after time $t + \nu$ will remain there until it gets p_i's $exit_s$ message. Its

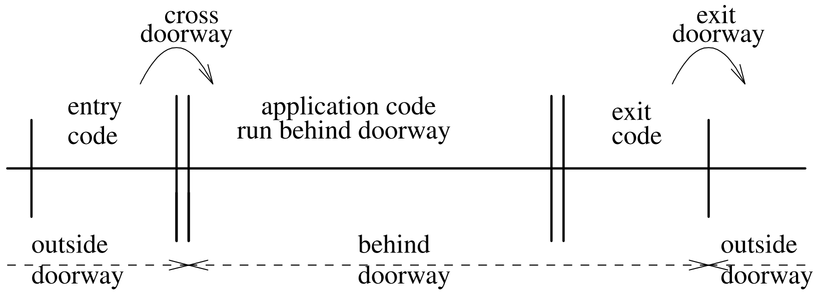

Figure 1: Schematic view of a doorway.

```
Synchronous doorway implementation:
Initially, $L[j] = exit_s$ for all $j \in N \cup \{i\}$
  Entry:
      wait until $\langle \forall j : j \in N : L[j] = exit_s \rangle$;
      broadcast $cross_s$ message;
      $L[i] := cross_s$;
  Exit:
      broadcast $exit_s$ message;
      $L[i] := exit_s$;

Handling link creations:
  LinkUp($j$) indication while static:
      $L[j] := exit_s$
      send $L[i]$ to $j$
  LinkUp($j$) indication while moving:
      broadcast $exit_s$ message;
      $L[i] := exit_s$
      wait until $L[j]$ is received
```

```
Asynchronous doorway implementation:
Initially, $L[j] = exit_a$ for all $j \in N \cup \{i\}$
  Entry:
      $\langle \forall j : j \in N$ : wait until $L[j] = exit_a \rangle$;
      broadcast $cross_a$ message;
      $L[i] := cross_a$;
  Exit:
      broadcast $exit_a$ message;
      $L[i] := exit_a$;

Handling link creations:
  LinkUp($j$) indication while static:
      $L[j] := exit_a$
      send $L[i]$ to $j$
  LinkUp($j$) indication while moving:
      broadcast $exit_a$ message;
      $L[i] := exit_a$
      wait until $L[j]$ is received
```

Figure 2: Basic doorway implementations for p_i. N holds the set of current neighbors of p_i, $L[j]$ stores the last message that p_i received from p_j, initialized to $exit_s$ or $exit_a$ according to the type of the doorway. A new neighboring node is considered to be outside the doorway. When a node p_j moves, the LinkDown indication causes its neighbor p_i to remove j from its N set.

weakness lies in the fact that this doorway alone does not avoid starvation of a node in the entry code, as its neighbors may repeatedly cross the doorway, blocking a node waiting in the entry code. Consider node p_i having two neighbors p_j and p_k which do not have a link between them. If the time intervals of p_j and p_k being behind the synchronous doorway are interleaved such that p_j, p_k or both are behind the doorway, p_i will remain waiting forever in the entry code of the synchronous doorway.

A *double doorway* [9] is a synchronous doorway within an asynchronous one (Figure 3). 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 synchronous doorway, yet ensures that after a period of time no new neighbors cross the doorway and execute concurrently with a node behind the

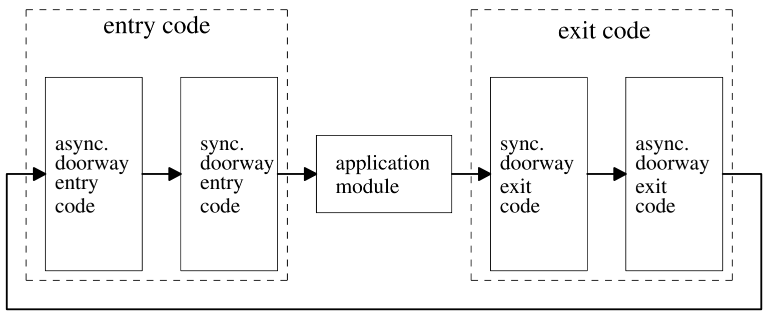

Figure 3: Double doorway.

doorway. This property is important for our first local mutual exclusion algorithm.

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

Lemma 1. *A node that starts executing the entry code of the double doorway at time t 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.*

Proof: Let p_i be a node that crosses the asynchronous doorway at time t_i. If none of its neighbors is behind the synchronous doorway, p_i crosses it immediately. Otherwise, it waits for each of its neighbors to exit the synchronous doorway. Any node that exits the asynchronous doorway after time $t_i+\nu$ cannot cross the asynchronous doorway again until p_i exits it. Since the synchronous doorway is enclosed in the asynchronous one, any node that exits the synchronous doorway after time $t_i + \nu$ cannot cross the synchronous doorway again until p_i exits it. As any new neighbor that arrives in p_i's neighborhood after t_i will not cross even the asynchronous doorway before p_i exits the double doorway (since it will find p_i behind the asynchronous doorway), we conclude that the number of neighbors that may delay p_i from crossing the synchronous doorway is limited by δ, the maximum number of neighbors of p_i at t_i. By the assumption, each neighbor p_k may delay p_i for at most T time units while behind the synchronous doorway, provided that no node in p_k's m-neighborhood fails. As a result, assuming no node in the $(m + 1)$-neighborhood of p_i fails, p_i crosses the synchronous doorway by time $t_i + O(\delta T)$.

Now, consider a node p_j that starts the entry code of the asynchronous doorway at time t_j. By the argument given in the previous paragraph, any neighbor p_i of p_j that is behind the asynchronous doorway, exits it by time $t_j + O(\delta T)$, provided no node fails in p_i's $(m + 1)$-neighborhood. Therefore, the waiting condition of the entry code of the asynchronous doorway holds for p_j by time $t_j + O(\delta T)$, provided no node fails in its $(m+2)$-neighborhood. As a result, by this time p_j crosses the asynchronous doorway, and after at most $O(\delta T)$ time units crosses the synchronous one, which means that it exits the double doorway by time $t_j + O(\delta T)$. □

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 4). The decision is taken by the application module

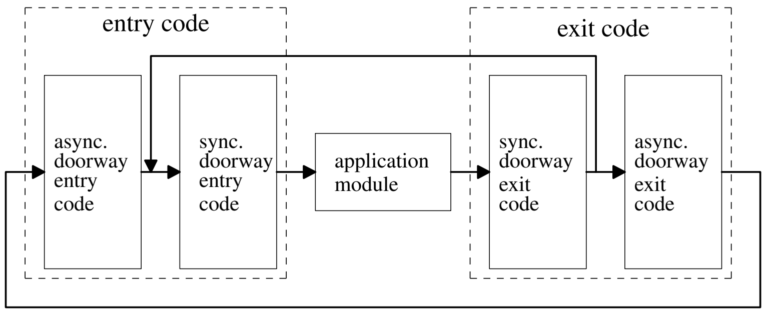

Figure 4: Double doorway with a return path. When a node exits the internal synchronous doorway, it may exit the double doorway or return to the entry code of the internal synchronous doorway.

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. When the return path is not taken, this doorway is identical to a double doorway.

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

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

Proof: Let p_i be a node that crosses the asynchronous doorway at time t_i. If none of its neighbors is behind the synchronous doorway, p_i crosses the synchronous doorway immediately. Otherwise, consider a neighbor p_j that is behind the synchronous doorway at time t_i. Node p_j exits the synchronous doorway by time $t_i + T$, if no node in its m-neighborhood fails. It may then return to the entry code of the synchronous doorway and enter it again before p_i crosses it. The number of executions of the module behind the synchronous doorway is at most R and hence p_j may delay p_i from crossing the synchronous doorway for at most $(T+\nu)R$ time units. In addition, p_i may cross the synchronous doorway and return to its entry code several times, which means that it may spend at most $(R-1)T$ time units behind the synchronous doorway before it enters it for the last time. Since any new neighbor that arrives at p_i's neighborhood after t_i will not cross the asynchronous doorway before p_i exits the double doorway, we conclude that the number of neighbors that may delay p_i from crossing the synchronous doorway is at most δ. Thus, p_i enters the synchronous doorway for the last time within $(T+\nu)R\delta + (R-1)T$ time units. Adding another T time units for executing the module inside the doorway, we get that p_i exits the double doorway by time $t_i + O(\delta TR)$, if no node in its $(m+1)$-neighborhood fails.

Similarly to the argument used in Lemma 1, consider a node p_j that starts the entry code of the asynchronous doorway at time t_j. By the same analysis, all neighbors of p_j that are behind the asynchronous doorway at time t_j, exit this doorway by time $t_j + O(\delta TR)$, provided no node in their $(m+1)$-neighborhood fails. Thus, p_j crosses the asynchronous doorway by time $t_j + O(\delta TR)$, provided no node in its $(m+2)$-neighborhood fails. Thus, p_j exits the double doorway with a return path by $O(\delta TR)$ time units after it starts executing its entry code. □

Chapter 5

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

Our first algorithm uses *forks* as a metaphor for a resource shared among 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 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. 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 and decreases the performance of the fork collection. 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 5).

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 range 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 relatively small range. We provide two implementations for this algorithm.

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

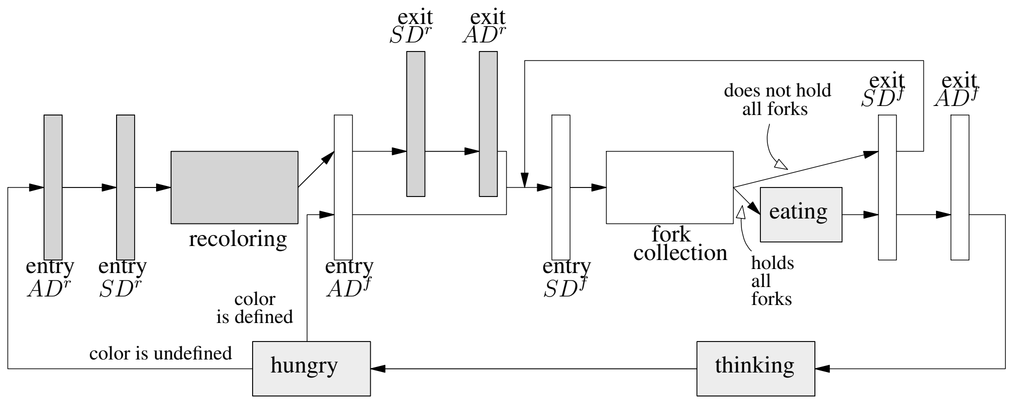

Figure 5: Schematic description of the first algorithm.

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 asynchronous 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 logical 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.

5.1 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

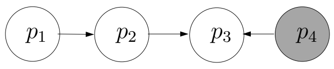

Figure 6: Example for a scenario that may increase the response time in the fork collection module due to nodes mobility. Edges are directed from nodes with larger colors to nodes with smaller colors.

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 6 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 holding 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 5 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.

5.2 Pseudo-code

The first local mutual exclusion algorithm appears in Algorithms 1–3. In the main module (Algorithm 1), which handles fork collection, each node p_i has the following local variables:

- N, set of IDs of neighboring nodes. Maintained by the lower level.
- $color[j]$, 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*.
- $at[j]$, boolean flag, indicating that p_i holds the fork shared with p_j; initially *true* when $ID[i] < ID[j]$.
- S, set of neighbors with suspended forks requests.

The code uses the macros *all-(low)-forks* to indicate that the node has all its (low) shared forks: *all-forks* holds when $at[j]$ is *true* for all $j \in N$, while *all-low-forks* holds when $at[j]$ is *true* for all $j \in N$, such that $color[j] < color[i]$.

Algorithm 1 Main module, code for p_i

```
 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 as indicated in $color[]$
 7:    broadcast $update\text{-}color(color[i])$ 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($j$)
```

Lines 1–4 describe the actions taken by a hungry node when it crosses the second double doorway. If the node already has all its forks, then it starts eating. Otherwise, if it has all its low forks (but not all its high forks), it requests the missing high forks. If it does not have all its low forks, then it requests the missing low forks.

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; 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

Algorithm 2 Wrapper code for the recoloring module, code for p_i

```
36: when $SD^r$ is crossed (and $state$ is *hungry*):
37:     $R := N$
38:     $color[i] := (-1) * $new-color()$ - 1$        // new-color() is presented in Section 5.4
39:     broadcast $update\text{-}color(color[i])$ 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:     $R := R \setminus \{j\}$
```

recoloring module picks colors in a range of a greater size. The new color is sent to all neighbors and delayed requests for forks are fulfilled (Lines 7–8). Finally, both fork-collection synchronous and asynchronous doorways are exited (Line 9).

Lines 10–16 handle a request for a fork. If the request comes from a high neighbor, it is fulfilled only if the recipient node does not hold all low forks or is outside the double doorway; otherwise, the request is suspended until the node eats or loses one of its low forks. A request from a low neighbor (Lines 13–15) is granted if the node does not yet hold all forks; otherwise, the request is suspended. If the request is granted, in addition to sending the requested fork, all suspended high forks are returned.

Lines 17–23 handle a fork message received from a neighbor. The message includes a boolean *flag*, which is set by the sender of the fork when the sender wants this fork back. If this fork causes the receiver to have all low forks, the receiver asks for missing high forks; the receiver suspends the request from the sender of the fork if *flag* is true. Otherwise, the receiver sends the fork back immediately if *flag* is true. A fork is not sent back and forth indefinitely, since *flag* is turned on only by a sender with lower priority (Line 31). Note that it is unnecessary to set *flag* to true if the request being granted is for a high fork: since the request is being granted, it follows that the node does not have all its low forks; if it ever gets all its low forks (again), it will explicitly ask for all missing high forks.

The recoloring module (Algorithm 2) presents the wrapper code for the coloring procedure that calculates new colors for nodes. The wrapper code introduces one variable, R, which holds a set of neighbors behind the doorway that are participating in the recoloring; it is initialized to N.

A node enters the recoloring module when it crosses the first double doorway. The new color is set to one less than the negative of the value returned from coloring procedure new-color() (Line 38), presented in two variations in Section 5.4. 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 rest of Algorithm 2 (Lines 40–43) provides code for handling recoloring-related messages received by or from nodes that are not participating in recoloring.

Algorithm 3 handles dynamic topology changes.

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

Algorithm 3 Link creation and failure, code for p_i

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

as the owner of the shared fork and initializes the color of the moving node to $\perp$ (undefined value); 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 arriving 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 6), then it exits the synchronous doorway and returns to its entry code (Lines 59–60).

We assume that a lower link protocol updates the node's neighbor set (N) according to notifications about link creations and failures. A node behind the SD^r doorway takes a special action only when an incident link fails (Line 61).

5.3 Correctness proof

We show that colors of nodes running the fork collection modules are legal, provided the recoloring module assigns legal colors to nodes. Then we proceed with the analysis of the fork collection module.

A key aspect in the analysis of the fork collection module is the notion of the "rank" of a node in a directed graph that reflects the priorities of competing nodes. (This notion is also used in our second algorithm, which uses boolean values instead of colors to indicate priorities.) Lemma 5 and Corollary 6 prove how the double doorway with a return path ensures that after some time, no nodes are added to this graph, so the rank of any node in the graph is bounded at this time. Lemma 8 states that if a hungry node remains static sufficiently long, then it starts eating by some predefined time, or its set of low forks does not change starting from some predefined time. Lemma 8 is proved by induction on the rank of the hungry node; Lemma 7 is the basis of the induction. Lemma 9 shows that the starvation situations cannot occur as long as no node in the 2-neighborhood of the hungry node fails.

Finally, we show that when the fork collection module is combined with the recoloring module and the doorways, the result is an algorithm with failure locality $O(f_{color})$ and response time in $O((T_{color} + \delta^2\Delta)\delta)$, where f_{color} and T_{color} are failure locality and response time of the recoloring module, respectively, and Δ is the maximal absolute value of a color produced by the recoloring module. After presenting two coloring procedures in Section 5.4 and proving that they assign legal colors to nodes, we derive the actual complexity properties of two versions of the first algorithm for local mutual exclusion in MANETs in Theorem 16 and Theorem 22, respectively.

We start the analysis by proving that the algorithm satisfies the local mutual exclusion property.

Lemma 3. *The first algorithm satisfies the local mutual exclusion property.*

Proof: A node must have all forks before accessing the critical section. When a link is created between two nodes, a single fork is created which is held by at most one node; the fork is destroyed upon destruction of the link. When a fork is sent to a neighbor, a corresponding at flag is set to *false*. Thus, $at_i[j]$ and $at_j[i]$ do not hold for any two neighboring nodes p_i and p_j simultaneously, implying local mutual exclusion. $\square$

We formulate an assumption on 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 4. *If p_i is behind the second double doorway in the time interval $[t_1, t_2]$, then for any neighbor p_j of p_i that is behind the same doorway at time $t \in [t_1, t_2]$, $color[i] \neq color[j]$.*

Proof: Assume, by a way of contradiction, that $color[j] = color[i]$ at some time in $[t_1, t_2]$. If $color[j] = color[i] \in [0..\delta]$, both p_i and p_j have not moved since the last time they picked a color upon exit from the critical section. This is a contradiction since these colors are chosen in exclusion.

Thus, $color[j] = color[i] < 0$, implying that both p_i and p_j picked the color in the recoloring module. Let t_i and t_j be the times when p_i and p_j started recoloring, respectively. Without loss of generality, assume that $t_i \leq t_j$. If p_j started recoloring when p_i was behind the first double doorway, then they pick different colors, by Assumption 1. Otherwise, p_i exits the first double doorway before p_j enters it. The doorways are interleaved so that p_i crosses AD^f before it exits AD^r and SD^r (Figure 5). The FIFO property of the links ensures that p_j does not cross AD^f until p_i exits it, contradicting the assumption that p_i and p_j are behind the second double doorway at the same time. □

For the rest of Section 5.3, the doorway refers to SD^f, unless specified otherwise. For any node p_i behind the doorway at a specific time, define a directed graph LG_i, consisting of hungry nodes that are behind the doorway, have equal or higher priority than p_i and are connected to p_i by a path of directed edges. Specifically:
— p_i is a node in LG_i.
— If p_j is a node in LG_i and its neighboring node p_k has a smaller color and is behind the doorway, then p_k is a node in LG_i and the edge is directed from p_j to p_k.

Note that the undirected version of LG_i is a subgraph of the system communication graph, which changes dynamically as nodes join and leave it by crossing and exiting the doorway and/or moving away. Additionally, LG_i is acyclic since colors are legal and drawn from a totally ordered set.

For Lemmas 5–9, fix a time t_i when a node p_i crosses the doorway. Recall that τ is the maximum eating time and ν is the maximum message delay.

Lemma 5. *If p_j joins LG_i at time $t > t_i + k * \nu$, where $k \geq 0$, and p_i is behind the doorway throughout the interval $[t_i, t]$, then the distance of p_j from p_i in LG_i at time t is at least $k + 1$.*

Proof: The proof is by induction on k.

Basis: $k = 0$. The only node at distance 0 from p_i at any time is p_i itself, and the claim trivially holds.

Induction step: $k > 0$. Suppose, by way of contradiction, that the distance between p_i and p_j in LG_i at time t is $\leq k$. Then p_j has a neighbor p_m in LG_i whose distance from p_i in LG_i is $\leq k - 1$. By the inductive hypothesis, p_m joins LG_i by time $t_i + (k - 1) * \nu$. When it joins LG_i, p_m sends a $cross_s$ message. If p_j receives that message, then it does so by time $t_i + k * \nu$. If p_j does not receive that message, then it must be a newly arrived neighbor of p_m, so it waits for p_m's doorway status (Line 53). In either case, p_j observes p_m as being behind the doorway. If p_m stays there until time $\leq t$, p_j cannot cross the doorway and thus cannot join LG_i during the interval $(t_i + k * \nu, t]$, which is a contradiction. If p_m exits the doorway before time t, p_j may cross the doorway, but it does not join LG_i, by definition. Thus, p_j cannot join LG_i after $t_i + k * \nu$, which is a contradiction. □

Let Δ be the maximal absolute value of a color produced by the recoloring module. A node sets its color either in the recoloring module or when exiting the critical section. Since colors on a directed path in LG_i are decreasing, the maximum path length (distance from p_i) is $\Delta + color[i]$, which implies:

Corollary 6. *If p_i is behind the doorway throughout the interval $[t_i, \xi]$, where $\xi > t_i + \nu * (\Delta + color[i])$, then no nodes join LG_i at any time $t \in (t_i + \nu * (\Delta + color[i]), \xi]$.*

Lemma 7. *If p_i is behind the doorway throughout the interval $[t, \xi]$, where $t \geq t_i$ and $\xi \geq t + \tau + 2\nu$, holds all low forks at time t and does not receive a request from a low neighbor in the interval $[t, \xi]$, then one of the following two conditions holds:*

1. *p_i starts eating by time $t + \tau + 2\nu$,*
2. *one of p_i's high neighbors fails in the interval $[t, \xi]$.*

Proof: If p_i holds all high forks at time t, it starts eating immediately, establishing Condition 1. Otherwise, consider p_j, a high neighbor of p_i, that is holding the shared fork at time t. By Lines 3 and 22, p_i sends a request for the missing fork by time t, which arrives at p_j by time $t + \nu$. If p_j has not failed, it responds with a fork, possibly after eating, by time $t + \nu + \tau$; the fork reaches p_i by time $t + 2\nu + \tau$ and p_i keeps the fork until it exits the doorway. Thus, if every high neighbor of p_i is not failed, then p_i collects all the forks and starts eating by time $t + 2\nu + \tau$, establishing Condition 1. Otherwise, some high neighbor p_k fails in the interval $[t, \xi]$ and does not respond with a fork, establishing Condition 2. □

Let T_i be the time of the first event after the last node joins LG_i until p_i exits the doorway. Consider LG_i at time T_i. The *rank* of a node p_j in LG_i, denoted $r_i(j)$, is the length of the longest path from p_j to any other node in LG_i at time T_i. Since LG_i is acyclic, the rank is well-defined for any node. A node with no outgoing edges has rank 0 (and highest priority) and the rank of any node is bounded by $\Delta + \delta$. Additionally, if p_j and p_k are two neighbors in LG_i with an edge directed from p_j to p_k, then $r_i(j) \geq r_i(k) + 1$.

To simplify the calculation of the response time, we also define for each node p_j in LG_j a function $f_j = r_i(j) * (2\tau + 4\nu) + 2\tau + 4\nu$.

Lemma 8. *If p_i is behind the doorway throughout the interval $[t_i, \xi]$, where $\xi \geq T_i + f_i$, then one of the following two conditions holds:*

1. *p_i starts eating by time $T_i + f_i$,*
2. *all predicates $at_i[k]$ do not change in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, where p_k ranges over the low neighbors of p_i.*

Proof: The proof is by induction on $r_i(i)$, the rank of p_i in LG_i.

Base: $r_i(i) = 0$. In this case, p_i does not have a hungry low neighbor behind the doorway. Node p_i sends a request for its low forks when it crosses the doorway. The low neighbors respond with the forks, which reach p_i by time $T_i + \tau + 2\nu$. These forks are not sent back until p_i exits the doorway, since p_i's low neighbors do not cross the doorway after time T_i (the time after which no new nodes are added to LG_i). If one of the low neighbors of p_i fails while in possession of the fork shared with p_i, p_i never gets this fork, thus Condition 2 is established. Otherwise, Lemma 7 can be applied to p_i, with $t = T_i + \tau + 2\nu$, because p_i has all low forks and does not receive a request from a low neighbor throughout $[t, \xi]$, since the interval is after T_i. Thus, either p_i starts eating by time $T_i+2\tau+4\nu$, establishing Condition 1, or p_i remains behind the doorway throughout the interval $[t_i, \xi]$ while holding all low forks in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, establishing Condition 2.

Induction step: Assume that the induction hypothesis holds for all nodes p_k with rank less than $r_i(i)$ in LG_k. Let p_j be some low neighbor of p_i. Informally, our goal is to show that if p_i does not start eating by time $T_i + f_i$, then the fork shared between p_i and p_j does not change ownership in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, implying that the predicate $at_i[j]$ does not change in this interval.

If p_j is outside the doorway at time T_i, it does not cross the doorway until ξ, since no new neighbors join LG_i in the interval $[T_i, \xi]$. Node p_i requests the fork when p_i crosses the doorway. This request reaches p_j by time $T_i+\nu$, and p_j responds with a fork which is received by p_i by time $T_i+2\nu$. The fork is owned by p_i until p_i leaves the doorway. Thus, the fork does not change ownership in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, as long as p_i does not start (and finish) eating.

So, assume p_j is behind the doorway at time T_i. We distinguish between whether p_j exits the doorway or not by time $T_i + f_i - \tau - 3\nu$.

Case 1: Node p_j exits the doorway by time $T_i + f_i - \tau - 3\nu$. This may happen if p_j finishes eating (Line 9), one of p_j's low neighbors moves holding a fork (Line 60), or p_j moves. In the first two cases, p_j sends all requested forks to its neighbors; thus, p_i gets the fork shared with p_j by time $T_i+f_i-\tau-2\nu$. Node p_j does not cross the synchronous doorway again until p_i exits it. Consequently, p_j does not ask for the fork back until time ξ or later; thus, the shared fork stays with p_i in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, or as long as p_i does not start (and finish) eating.

In the third case, when p_j moves, if p_i does not hold the fork when it receives the *LinkDown(j)* indication, p_i exits the doorway (by Line 60), contradicting the assumption that p_i is behind the doorway in the interval $[t_i, \xi]$. If p_i holds the fork when the *LinkDown(j)* indication is received, p_j is removed from the set of p_i's neighbors, ensuring that the fork does not return to p_j until time ξ or later.

Case 2: Node p_j is behind the doorway throughout the interval $[t_j, \xi']$, where $\xi' \geq T_i+f_i-\tau-3\nu$. Since p_j is a low neighbor of p_i and LG_j is a subgraph of LG_i, it follows that $T_j \leq T_i$, where T_j is the the time of the first event after the last node joins LG_j until p_j exits the doorway. From the definition it follows that $r_j(j) \leq r_i(i) - 1 = r - 1$, and thus $T_j + f_j \leq T_i + f_i - 2\tau - 4\nu$. Thus, $T_i + f_i - \tau - 3\nu > T_j + f_j$ and the induction hypothesis on p_j implies that one of the following two conditions holds:

Case 2.1: Node p_j starts eating by time $T_j + f_j$. Then node p_j finishes eating by time $T_j + f_j + \tau$ and sends the fork to p_i, which receives the fork by time $T_j + f_j + \tau + \nu < T_i + f_i - \tau - 2\nu$. Since p_i remains behind the doorway, p_j does not cross the doorway again until time ξ, thus this fork is owned by p_i in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, as long as p_i does not start (and finish) eating.

Case 2.2: All predicates $at_j[k]$ do not change in the interval $[T_j + f_j - \tau - 2\nu, \xi']$, where p_k ranges over the low neighbors of p_j. Thus, the last request for the fork from p_j to p_i until time ξ' is sent by time $T_j + f_j - \tau - 2\nu$, and reaches p_i by time $T_j + f_j - \tau - \nu$. If p_i starts eating by time $T_j + f_j - \tau - \nu$, Condition 1 is established. Otherwise, p_i responds with a fork message with *flag* turned on, which arrives at p_j by time $T_j + f_j - \tau$.

If by time $T_j + f_j - \tau - 2\nu$ node p_j holds all low forks, p_j does not return the fork to p_i. Node p_j does not move away while holding the fork before time ξ, since that would cause p_i to exit the doorway before time ξ, contradiction. Since node p_j does not start eating by time $T_j + f_j$, it remains behind the doorway throughout the interval $[t_j, \xi]$, while holding the fork shared with p_i throughout the interval $[T_j + f_j - \tau, \xi]$. Since $T_j + f_j - \tau < T_i + f_i - \tau - 2\nu$, $at_i[j]$ remains *false* throughout the interval $[T_i + f_i - \tau - 2\nu, \xi]$.

If by time $T_j + f_j - \tau - 2\nu$ node p_j does not hold all low forks and it has not failed yet, p_j returns the fork back to p_i, since *flag* is turned on (if p_j fails, it never responds with the fork; thus, $at_i[j]$ is *false* throughout the interval $[T_i + f_i - \tau - 2\nu, \xi]$). This fork reaches p_i by time $T_j + f_j - \tau + \nu < T_i + f_i - \tau - 2\nu$. Node p_j remains behind the doorway without holding all low forks. Consequently, p_j does not issue a new request for the fork shared with p_i after time $T_j + f_j - \tau - 2\nu$. Note that if p_j moves away or exits the doorway due to a movement of some of its low neighbors holding a fork after time $T_j + f_j - \tau + \nu$ (the time by which p_i receives the fork from p_j), it would not cause p_i to send the fork back to p_j. Thus, $at_i[j]$ remains *true* throughout the interval $[T_i + f_i - \tau - 2\nu, \xi]$. □

Using the previous lemma, we prove that if a node p_i is behind the doorway sufficiently long and no nodes fail in its 2-neighborhood, then p_i will enter the critical section.

Lemma 9. *If p_i is behind the doorway throughput the interval $[t_i, T_i + f_i]$ and no nodes within a distance of 2 from p_i fail, then p_i starts eating by time $T_i + f_i$.*

Proof: Apply Lemma 8 on node p_i with some $\xi \geq T_i + f_i$. If Condition 1 holds, we are done. Thus, assume Condition 2 holds. If p_i holds all low forks by time $t = T_i + f_i - \tau - 2\nu$, Lemma 7 for time t and the fact that no high neighbor of p_i fails imply that p_i starts eating by time $T_i + f_i$.

Thus, assume p_i misses some of its low forks by time t and let p_j be a low neighbor of p_i, such that $at_i[j]$ is *false* in the interval $[T_i + f_i - \tau - 2\nu, \xi]$. If p_j is not behind the doorway at time $t_i + \nu$, then, by the condition in the entry code of the doorway, p_j does not cross the doorway until time ξ. By Line 4, p_i requests the fork shared with p_j by time t_i; this fork arrives to p_i by time $t_i + 2\nu$ and remains owned by p_i until time ξ, contradicting the assumption that $at_i[j]$ is *false* in the interval $[T_i + f_i - \tau - 2\nu, \xi]$.

Thus, p_j is behind the doorway at time $t_i + \nu$. Node p_j does not move away while holding the fork until ξ, since this would cause p_i to exit the doorway, contradicting the assumption that p_i is behind the doorway until ξ. If p_j starts eating or exits the doorway because one of its low neighbors moves by time $T_j + f_j$, the shared fork reaches p_i by time $T_j + f_j + \tau + \nu < T_i + f_i - \tau - 2\nu$. By the definition of T_i, p_j does not cross the doorway again until time ξ, thus the shared fork remains with p_i in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, contradicting the assumption that $at_i[j]$ is *false* in this interval.

Therefore, p_j is behind the doorway throughout the interval $[t_j, \xi']$ for some ξ', where $\xi' \geq T_j + f_j$. Apply Lemma 8 on p_j. Since p_j does not start eating by time $T_j + f_j$, Condition 2 must hold. If p_j holds all low forks in the interval $[T_j + f_j - \tau - 2\nu, \xi']$, Lemma 7 and the fact that no neighbors of p_j fail imply that p_j starts eating by time $T_j + f_j$, which is a contradiction.

Thus, p_j misses one of its low forks in the interval $[T_j + f_j - \tau - 2\nu, \xi']$. Thus, the last request by p_j until ξ' for the fork shared with p_i has to be sent by time $T_j + f_j - \tau - 2\nu$. The fork returns to p_j

with *flag* turned on by time $T_j + f_j - \tau$. This fork is sent back to p_i since *all-low-forks* is *false* for p_j. Thus, p_i receives the fork by time $T_j + f_j - \tau + \nu < T_i + f_i - \tau - 2\nu$. Node p_j leaves the doorway before time ξ, i.e., $\xi' < \xi$, only if one of p_j's low neighbors moves while holding the shared fork, or if p_j itself moves (p_j does not eat, since it misses some forks). In both cases, p_i does not return the fork until ξ. The same is true if p_j remains behind the doorway (at least) until time ξ, i.e., $\xi' \geq \xi$, since p_j does not issue a new request. Thus, $at_i[j]$ is *true* in the interval $[T_i + f_i - \tau - 2\nu, \xi]$, contradicting the assumption that $at_i[j]$ is *false* in this interval. □

Denote the time complexity of the recoloring module, that is the number of time units from the time a node crosses the first doorway till it finds a new color, by T_{color} and its failure locality by f_{color}. Recall that Δ is the maximal absolute value of a color produced by the recoloring module.

Lemma 10. *The first algorithm has failure locality* $\max(f_{color}, 4) + 2$ *and response time* $O((T_{color} + \delta^2(\Delta + \delta))\delta)$.

Proof: We show first that if there are no failures in the $(\max(f_{color}, 4) + 2)$-neighborhood of a hungry node, the node will get eventually into critical section, and then we bound the time that is required until the node enters the critical section.

To bound the failure locality, note that by Lemma 9, a node executing the fork collection module starts eating if no nodes fail in its 2-neighborhood. Thus, by Lemma 2, the fork collection module enclosed in the second double doorway has failure locality 4. Since part of the second double doorway is included in the first double doorway and this part is crossed only after a node finishes the recoloring module, a node that crosses the first double doorway exits it if no node in its $\max(f_{color}, 4)$-neighborhood fails. Lemma 1 implies that the entire algorithm has failure locality $\max(f_{color}, 4) + 2$.

To bound the response time, suppose node p_i becomes hungry at time t, remains static until it eats and no node fails in its $(\max(f_{color}, 4) + 2)$-neighborhood. The complete algorithm is structured as the execution of one double doorway (entry code, enclosed module, and exit code) and the execution of one double doorway with a return path, with the beginning of the second double doorway overlapping the end of the first one.

By Lemma 1, node p_i exits the first double doorway by time $t + O(T\delta)$, where T is the time taken by the module enclosed by the double doorway. The enclosed module consists of the recoloring algorithm followed by the entry to AD^f. The time for the the recoloring algorithm is T_{color} and the time for the entry to AD^f is at most the time it takes a node to execute the entire second double doorway. We show below that this time is $O(\delta^2(\Delta + \delta))$. Thus p_i completes the first double doorway by time $t + O((T_{color} + \delta^2(\Delta + \delta))\delta)$.

By Lemma 2, the time taken to execute the second double doorway with a return path is $O(\delta RT)$, where R is the maximal number of times the entry code of SD^f is executed by a node before it exits the double doorway, and T is the time taken by the module enclosed by the double doorway. The enclosed module consists of the fork collection module followed by the critical section.

Recall that t_i is the time when p_i crosses the second double doorway. Since no nodes in the 2-neighborhood of p_i fail, if p_i remains behind this doorway until time $T_i + f_i$, then by Lemma 9, p_i must start eating by that time. Since $r_i(i) \leq \Delta + \delta$, the definition of f_i ensures that $f_i = O(\Delta + \delta)$ and, by Corollary 6, the definition of T_i ensures that $T_i = t_i + O(\Delta + \delta)$. Thus, p_i starts eating by time $t_i + O(\Delta + \delta)$.

As a result, p_i leaves the synchronous doorway by time $t_i + O(\Delta + \delta)$ with or without eating. The latter could happen if a low neighbor holding a fork moves; in this case p_i will return to the entry of the synchronous doorway again using the return path. Node p_i can traverse the return path at most δ times, once for each node that is a low neighbor of p_i at time t_i.

Consequently, substituting R by δ and T by $O(\Delta + \delta)$, we conclude that p_i completes the second double doorway by $O(\delta^2(\Delta + \delta + \tau)) = O(\delta^2(\Delta + \delta))$ time units, and starts eating by time $t + O((T_{color} + \delta^2(\Delta + \delta))\delta)$. □

In the static setting, the return path of the second double doorway is never taken, that is, the maximal number of times each node executes the entry code of SD^f is 1. Thus, the same proof implies the following result:

Lemma 11. *When there are no changes in the topology, the first algorithm has response time* $O((T_{color} + \delta(\Delta + \delta))\delta)$.

Moreover, it worth noting that after all nodes in the static system set a color upon the exit from the critical section (in Line 6) in the range $[0, ..., \delta]$, the recoloring module is never run again. Thus, the response time in this special case becomes $O(\delta^2)$, as in the algorithm of Choy and Singh [9].

5.4 The coloring procedures

We present two versions of the coloring procedure `new-color()` used by the wrapper code in Algorithm 2, demonstrating a trade-off between a more practical implementation and better (theoretical) complexity properties. As can be seen from Lemma 10, 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 sub-graph and picks a new color in a greedy manner, depending on its neighbors. This procedure has poor failure locality and response time that are both $O(n)$, and produces colors in a range of $O(\delta)$, but it does not assume nodes have a knowledge of n and δ, as the second coloring procedure does.

The second coloring procedure is more sophisticated, implementing an adaptation of the synchronous fast graph coloring algorithm of Linial [22]. This procedure 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. This approach achieves failure locality and response time in $O(\log^* n)$ at the price of increased combinatorial complexity, heavy pre-computation and the assumption of knowing n and δ.

Algorithm 4 Greedy coloring, code for p_i

```
62: new-color():
63:     $G := \emptyset$
64:     do
65:         send $\langle G, false \rangle$ to all $j \in R$
66:         wait until $\langle G_j, finished_j \rangle$ is received from all $j \in R$
67:         $G := G \cup G_j \cup \{(i,j)\}$ for all $j \in R$
68:     while ($G$ is changed) $\wedge$ ($finished_j = false$ for all $j \in R$) $\wedge$ ($R \neq \emptyset$)
69:     if ($R = \emptyset$) then    return 0
70:     else
71:         send $\langle G, true \rangle$ to all $j \in R$
72:         return calc-greedy-color($G$)
```

5.4.1 Greedy coloring

The pseudo-code for the greedy coloring procedure is presented in Algorithm 4. There, in addition to local variables mentioned in Section 5.2, each node p_i maintains a graph, represented as a set G of edges between nodes performing recoloring concurrently; it is initialized to $\emptyset$.

In each iteration of the loop in Lines 64–68, p_i exchanges the graph G with its neighbors and updates the graph preparing it for the next iteration. The loop ends when one of the following conditions holds (Line 68): (1) no more new edges are received, or (2) some neighbor p_j reports that it has finished its loop, that is, in the last message from p_j, the flag $finished_j$ is *true*, or (3) p_i recognizes that none of its neighbors is running the recoloring concurrently.

If p_i finishes the loop for the last reason, it decides on a new color immediately (Line 69) and returns from the procedure. Otherwise, p_i sends the final graph G to its neighbors with $finished$ flag turned on (Line 71) and runs the greedy coloring algorithm on G (Line 72), where p_i chooses the smallest possible color for itself, traversing the graph in some predefined order (e.g., DFS starting from a node with smallest ID).

We show next that despite node movements, two neighbors p_i and p_j running the greedy coloring procedure concurrently, finish the loop with the same graph G. This immediately implies that the recoloring module based on this procedure satisfies Assumption 1, which states that p_i and p_j receive different colors.

Let $T_i(k)$ be the time p_i finishes iteration k of the loop. Let $H(t')$ be the subgraph of the communication graph consisting of nodes doing recoloring at time t' (i.e., nodes that crossed the first double doorway and did not start the entry code of the second double doorway). Consider some node p_i and two neighboring nodes p_1 and p_2 in $H(t')$. Our algorithm allows an edge $e = (p_1, p_2)$ to be in G_i at the end of the last iteration of p_i even if p_1, p_2 or any other node on the shortest path between p_i and e in $H(t')$ move. We define the distance between p_i and e taking care of the fact that nodes in $H(t')$ may move by considering the information that intermediate nodes on the path between p_i and e in $H(t')$ have already collected.

Formally, the *distance* of an edge e from a node p_i at time $t \geq t'$ is defined as

$\min_{p_j \in H(t)}[d(p_i, p_j, H(t)) + d(p_j, e, G_j(t))]$, where $d(p_i, p_j, H(t))$ is the distance between p_i and p_j in $H(t)$, and $d(p_j, e, G_j(t))$ is the length of the shortest path from p_j in G_j at time t which includes the edge e.

Lemma 12. *At the end of iteration k, G_i contains exactly the edges at distance k or less from p_i at time $T_i(k)$, provided no node fails in the k-neighborhood of p_i.*

Proof: The proof is by induction on the number of iterations.

Base: Assume, by way of contradiction, that p_i receives a non-empty graph from its neighbor p_j in p_i's first iteration. Thus, p_i crosses the doorway after p_j sends its (empty) set on the first iteration. Thus, by Line 43, p_i responds with NACK on the message sent by p_j in p_j's first iteration. Thus, p_i cannot cross the doorway until p_j exits it, since by FIFO property of the links, p_i receives $cross_s$ message from p_j before p_i responds with NACK. Thus, p_i cannot receive a message from p_j on Line 66, which is a contradiction.

As a result, in the first iteration, p_i receives only empty graphs from all neighbors running the recoloring, Thus, p_i constructs G_i to be a set of edges between p_i and all these neighbors. That is, at $T_i(1)$, G_i holds all edges at distance 1 from p_i. An edge at distance larger than 1 does not appear in G_i.

Induction step: Assume that the claim holds for iteration k or less. Let p_i finish iteration $k+1$ and consider an edge e at distance $k+1$ from p_i at time $T_i(k+1)$. Let p_j be a neighbor of p_i such that the distance between p_j and e is k. Node p_j does not finish the loop before executing iteration k; otherwise, p_i would not execute $k+1$ iterations. Applying the induction hypothesis on p_j, we get that G_j contains e at the end of the iteration k. By Line 65 or 71 (depending if p_j finishes the loop after k iterations or more), p_j sends e to p_i, which reaches p_i by the end of the iteration $k+1$.

Assume that in iteration $k+1$, p_i receives an edge at distance $> k+1$. Let p_m be a neighbor that sent that edge to p_i. By definition, that edge is at distance $> k$ from p_m and was received by p_m at iteration k, contradicting the induction hypothesis. □

Lemma 13. *If p_i executes the loop in the time interval $[t_1, t_2]$, then the number of iterations executed by p_i is at least the maximum distance of any edge from p_i at time t_2.*

Proof: Let e be an edge at the maximal distance m from p_i at time t_2 and assume that p_i finishes the loop after m' iterations, where $m' < m$. By Line 68, the loop ends only if G_i is not changed, one of p_i's neighbors has finished the loop, or the set R_i becomes empty, meaning that all neighbors running recoloring with p_i moved. Let $p_j = p_i$ if G_i is not changed or R_i is empty at the end of iteration m'; otherwise, let p_j be a node at distance k from p_i that finishes the loop after $m' - k$ iterations since its graph G_j does not change. By simple induction on the distance between p_i and p_j, it can be easily shown that if p_i receives a message with $finish = true$, then there is a node at distance k from p_i that finishes the loop after $m' - k$ since its graph G does not change; denote this node as p_j. The last change in G_j occurs at the end of iteration $m' - k - 1$.

By definition, the distance between p_j and e at time $T_j(m' - k)$ is at least $m - k$. Let $p_j = p_0, p_1, ..., p_{m-k}$ be the prefix of the path between p_j and e at time $T_j(m' - k)$, as required by the definition of the distance. By Lemma 12, at the end of each iteration l, $l \leq m' - k < m - k$, the edge at distance l on the above path is contained in G_j; additionally, that edge cannot be added to G_j at the end of an earlier iteration. As a result, the graph G_j is constantly changed at the end of each iteration l, including the last iteration $m' - k$, which contradicts the way p_j was selected. □

Lemma 14. *The recoloring module with the coloring procedure in Algorithm 4 satisfies Assumption 1.*

Proof: Consider a node p_j that starts the recoloring module when its neighbor p_i is behind the first double doorway. Let t_i and t_j be the time p_i and p_j finish the loop, respectively. Assume, without loss of generality, that $t_i \leq t_j$. We show that $G_i = G_j$ at the time t_j, i.e., both neighbors finish the loop with the same graph G. Since p_i and p_j run the greedy color calculation on the same graph, they pick different colors.

Consider an edge $e \in G_i$. Since p_j is a neighbor of p_i, there exists a path at time $T_j(k)$, where k is the last iteration of p_j, between p_j and e, as required by the definition of the distance. Denote the length of that path by m. By Lemma 13, p_j finishes after at least m iterations, that is $k \geq m$. By Lemma 12, at the end of iteration m, G_j must include all edges at distance m from p_j, including e. Thus, $e \in G_j$ when p_j completes the loop. Thus, $G_i \subseteq G_j$. Similarly, $G_j \subseteq G_i$, so $G_i = G_j$. □

Lemma 15. *The recoloring module with the coloring procedure in Algorithm 4 finds a color in the range of $[0, ..., \delta]$ for a static node p_i in $O(n)$ time units from the time it crosses the first double doorway, provided that no node within a distance of n from it fails.*

Proof: Let p_i be a node crossing the first double doorway at time t. The distance of any edge from p_i at any time $t' \geq t$ is bounded by n, thus Lemma 12 implies that no new edge will be added to G_i at the end of iteration $n + 1$. Therefore, p_i runs at most $n + 1$ iterations. Provided none of the nodes running the recoloring module fails, each iteration takes at most 2ν time units; thus p_i finishes the loop and calculates its new color by time $t + O(n)$. The result of the greedy algorithm is a color in the range of $[0, 1, ..., \delta]$. □

Substituting $T_{color} = O(n)$, $f_{color} = n$ and $\Delta = \delta + 1$ in Lemmas 10 and 11, we get the next two theorems:

Theorem 16. *When run with the coloring procedure in Algorithm 4, the first algorithm satisfies the local mutual exclusion property in a MANET, has failure locality n and response time $O((n + \delta^3)\delta)$.*

Theorem 17. *When run with the coloring procedure in Algorithm 4 in the static setting, the first algorithm has response time $O((n + \delta^2)\delta)$.*

5.4.2 More efficient coloring

The recoloring module based on the simple greedy coloring procedure has failure locality that depends linearly on the number of nodes in the system. Theoretically, this means that a failure of one node may block the whole system, no matter how far other nodes are from the failed one. The following scenario shows that the failure locality of the recoloring module with the greedy coloring procedure can be as bad as n: all nodes in the system start running the recoloring simultaneously, and one of them fails in the first iteration, before sending its message in Line 65. Thus, all nodes at distance 1 from the failed node will be blocked in their first iteration in Line 66, all nodes at distance 2 will be blocked in their second iteration, and so on. Such scenario is increasingly likely to occur if nodes are moving frequently, thus performing the recoloring very often.

As a result, we propose an alternative coloring procedure, which has much better failure locality $O(\log^* n)$ and guarantees that the number of colors is proportional to $O(\delta^2)$. It does so by running, behind a double doorway, an adaptation of the fast graph coloring algorithm proposed by Linial [22] for synchronous and static networks, where all nodes start coloring simultaneously. This iterative algorithm assumes knowledge of n and δ and relies on the following combinatorial result:

Theorem 18 (Erdõs, Frankl and Füredi [14])**.** *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 theorem guarantees the existence of a set of subsets of a relatively small range of numbers, where each subset contains a number that does not appear in the union of any constant number of other subsets. 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. The final chosen colors are legal and in the range of size $O(\delta^2)$.

The existence of these sets is proved in [14] by probabilistic arguments, without constructing them explicitly. They can be found through an exhaustive search performed locally by each node. Since the sets depend only on the number of nodes n and the maximum degree δ, we can assume that all nodes are initialized with the same sets.

The pseudo-code for the second coloring procedure is presented in Algorithm 5. In addition to the local variables mentioned in Section 5.2, this procedure uses the following variables:

- ph, counts number of executed phases; initialized to 0.
- *temp-color*, temporary color chosen at the end of the previous iteration; initialized to ID of the node.
- CI, set of temporary colors of all nodes in R, in the current phase (recall that R holds the set of neighbors behind the doorway that are participating in the recoloring).

In each iteration of the loop in Lines 65-70, p_i sends its current temporary color and receives temporary colors of its neighbors which are also running the recoloring. Then it sets *temp-color* to some value in F_i that does not appear in $\bigcup_{p_j \in R} F_j$ (Line 68), where all sets $F_1, F_2, ..., F_n$ are locally precomputed.

The loop finishes after $\log^* n$ iterations or if p_i recognizes that none of its neighbors is running the recoloring concurrently. In the first case, the returned value is the temporary color calculated at the last iteration (Line 72), while in the latter case, zero is returned (Line 71).

The next lemma follows from the FIFO property of the links.

Algorithm 5 More efficient coloring, code for p_i

```
62: new-color():
63:     ph := 0
64:     temp-color := ID
65:     do:
66:         send temp-color msg to all $j \in R$
67:         wait until temp-color msg is received from all $j \in R$ and stored to $CI$
68:         temp-color := calc-new-color($R, CI$)          // using Linial's scheme
69:         $ph := ph + 1$
70:     while $(ph < \log^* n) \wedge (R \neq \emptyset)$
71:     if $(R = \emptyset)$ then    return 0
72:     else    return temp-color
```

Lemma 19. *The recoloring module with the coloring procedure in Algorithm 5 satisfies Assumption 1.*

Proof: Assume, by way of contradiction, that two neighboring nodes, p_i and p_j, set their *temp-color* to the same value at the end of iteration k. We show that at any time t, if two neighbors p_i and p_j are behind the first double doorway, then $i \in R_j$ and $j \in R_i$. This implies that p_i and p_j use the same subsets F_i and F_j in iteration k. Thus, if p_j sets *temp-color* to $x \in F_j$ it contradicts the fact that p_i sets *temp-color* to $x \in F_i \setminus F_j$.

Let t_i and t_j be the times when p_i and p_j crossed the first double doorway, respectively. Without loss of generality, assume that $t_i \leq t_j$. If p_j is not in N_i when p_i crosses this doorway, then p_j does not cross this doorway until p_i exits it, which is a contradiction. Thus, by Line 37, j is in R_i, and it is removed from this set only if p_j moves away or if p_j sends NACK message in Line 41. In the former case, p_i and p_j would not be neighbors anymore. In the latter case, p_j is outside the doorway and, by FIFO properties of the links, p_j receives the $cross_s$ message from p_i before getting the first *temp-color* message from p_i. Thus, p_j remains outside the doorway until p_i exits it, which is a contradiction. □

The next lemma applies to the first iteration of the loop in the coloring procedure.

Lemma 20. *If a node p_i crosses the first double doorway at time t, it picks a new temporary color in the range $\lceil 5\delta^2 \log n \rceil$ within $t + 2\nu$ time units, provided no node in the 1-neighborhood of p_i fails.*

Proof: The synchronous doorway ensures that every neighbor p_j of p_i enters the doorway by time $t + \nu$ and p_i receives p_j's color or NACK message by time $t + 2\nu$, since p_j does not fail. If all the neighbors respond with NACK messages, p_i picks color 0 and finishes (by Line 71).

All nodes are initialized with the same n sets, $F_1, F_2, \ldots, F_n$, that satisfy the hypothesis of Theorem 18 and R holds the set of neighbors of p_i participating in the recoloring. By the theorem, $F_i \not\subseteq \bigcup_{p_j \in R} F_j$, and hence p_i can set *temp-color* to some $x \in F_i \setminus (\bigcup_{p_j \in R} F_j)$. Clearly, $x \leq \lceil 5\delta^2 \log n \rceil$. □

Induction on the number of iterations implies:

Lemma 21. *The recoloring module with the coloring procedure in Algorithm 5 finds a color in the range of $O(\delta^2)$ for a static node p_i in $O(\log^* n)$ time units from the time it crosses the first double doorway, provided that no node within a distance of $\log^* n$ from it fails.*

Substituting $T_{color} = O(\log^* n)$, $f_{color} = \log^* n$ and $\Delta = O(\delta^2)$ in Lemmas 10 and 11, we get the next two results:

Theorem 22. *When run with the coloring procedure in Algorithm 5, the first algorithm satisfies the local mutual exclusion property in a MANET, has failure locality $\max(\log^* n, 4) + 2$ and response time $O((\log^* n + \delta^4)\delta)$.*

Theorem 23. *When run with the coloring procedure in Algorithm 5 in the static setting, the first algorithm has response time $O((\log^* n + \delta^3)\delta)$.*

Chapter 6

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 one 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, the algorithm 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 module without doorways or colors. The idea is to use dynamic priorities which change when nodes exit their critical sections [37]. 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.

Although this algorithm implements a modified mechanism to collect forks, the correctness proof relies on the tools used for Algorithm 1, which include the adapted definition of LG graph and of the rank of a node.

6.1 Pseudo-code

The pseudo-code appears in Algorithms 6 and 7. The algorithm uses the same set of variables as Algorithm 1, except the array $colors$, which is replaced by an array called $higher$. Initially, $higher_i[j]$ is $true$ if $ID[i] < ID[j]$.

When a node becomes hungry, it sends a *notification* message to its neighbors. A node p_j that receives a notification from p_i, replies with a special *switch* message (and sets $higher_j[i]$ to *true*) only if it is thinking and $higher_j[i] =$ *false*. Otherwise, it discards the message. The switch message causes its receiver to set the value of the $higher$ flag corresponding to the sender of the message to *false*. 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. By sending a switch message, p_j lowers its priority.

Next, a node starts the fork collection module with the following changes relative to Algorithm 1: $higher[j]$ is used in every place where $color[i]$ and $color[j]$ are compared in Algorithm 1, that is, $color[i] < color[j]$ is replaced by $\neg higher[j]$ and, analogously, $color[i] > color[j]$ is replaced by $higher_i[j]$. Lines 6 and 7, handling the choice of a new color for a node, are omitted since the nodes are not colored. The switch message is sent by p_i when it exits the critical section to lower p_i's priority regarding all its neighbors. The code handling link creation and failure is simpler, since doorways (and the associated return path) are not used.

6.2 Correctness proof

The analysis of this algorithm is also based on a definition of a directed graph (LG) and the notion of the rank, that reflects the priorities of competing nodes. Before providing a definition for LG, we define a more general graph G constructed from the communication graph with edges directed from a node with a lower priority to its neighbor with a higher priority. The graph G is used to show the acyclicity of LG, which is required for proper definition of the rank of a node.

Formally, for any two neighbors p_i and p_j in G, the edge is directed from p_i to p_j if and only if (1) $higher_j[i] = false$ or (2) both $higher_i[j]$ and $higher_j[i]$ are $true$ and there is a switch message in transit from p_i to p_j. It can be seen from the algorithm that the direction of an edge between two neighbors p_i and p_j is well-defined. Initially, exactly one of $higher_j[i]$ and $higher_i[j]$ is $false$, and similarly when a link is created. During an execution, $higher_i[j]$ is set to $false$ only when a switch message is received, but this happens only when p_j sets $higher_j[i]$ to $true$. Thus, at most one of $higher_j[i]$ and $higher_i[j]$ is $false$, and both are $true$ only while a switch message is in transit between p_i and p_j.

We prove that G is acyclic by using the argument of link reversal technique [15]. The acyclicity of G is clear when the algorithm is initialized, and can be shown to be preserved in one of the three possible ways the direction of an edge may change in Algorithm 6: (1) a node finishes eating, (2) a node establishes a connection with a new neighbor, or (3) a node gets a notification message from one of its neighbors with lower priority. In all three cases, all (incoming) edges incident to a node under consideration are reversed and a cycle cannot be created.

Lemma 24. *G is an acyclic directed graph in any state of the system.*

For any hungry node p_i, the definition of the graph LG_i is modified to be based on the $higher$ array. Now, a hungry node p_k is in LG_i if it has a neighboring node p_j in LG_i and one of the two conditions holds: (1) $higher_k[j] = false$ or (2) both $higher_j[k]$ and $higher_k[j]$ are $true$ and there is a switch message in transit from p_j to p_k. In this case, the edge is directed from p_j to p_k. Since LG_i is a sub-graph of the hungry nodes in G, its acyclicity follows immediately from Lemma 24

Using the same definition of the *rank* as before, we observe that here $r_i(j) \leq n - 1$ for any node p_j in LG_i. Consequently, denoting by t_i the time when p_i becomes hungry, Corollary 6 holds for

Algorithm 6 Main module, code for p_i

```
when $state$ is set to $hungry$:
    send $notification$ msg to all $j \in N$
    **if** all-forks **then** $state := eating$
    **if** all-low-forks **then** request-high-forks()
    **else** request-low-forks()

when $state$ is set to $thinking$:
    **for each** $j$ s.t. $\neg higher[j]$ **do**:
        send $switch$ msg to $p_j$;    $higher[j] := true$
    **for each** $j$ s.t. $j \in S$ **do**: send-fork($j$)

when $req$ msg is received from $j$ and $at[j]$:
    **if** ($\neg higher[j] \wedge \neg$all-low-forks) **then** send-fork($j$)
    **else if** ($higher[j] \wedge \neg$all-forks) **then**
        send-fork($j$);   release-high-forks()
    **else** $S := S \cup \{j\}$

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

when $notification$ msg is received from $j$:
    **if** (($state = thinking$) $\wedge \neg higher[j]$) **then**
        **for each** $k$ s.t. $\neg higher[k]$ **do**:
            send $switch$ msg to $p_k$;    $higher[k] := true$

when $switch$ msg is received from $j$:
    $higher[j] := false$

request-low-forks():
    **for each** $j \in N$ s.t. ($higher[j] \wedge \neg at[j]$) **do**:
        send $req$ msg to $j$

request-high-forks():
    **for each** $j \in N$ s.t. ($\neg higher[j] \wedge \neg at[j]$) **do**:
        send $req$ msg to $j$

send-fork($j$):
    send $\langle fork, (higher[j] \wedge (state = hungry)) \rangle$ to $j$
    $at[j] := false$;    $S := S \setminus \{j\}$

release-high-forks():
    **for each** $j \in S$ s.t. ($\neg higher[j] \wedge at[j]$) **do**:
        send-fork($j$)
```

Algorithm 7 Link creation and failure, code for p_i

```
40: $LinkUp(j)$ indication arrives while static:
41:     $at[j]$ := true;    $higher[j] := false$
42: $LinkUp(j)$ indication arrives while moving:
43:     $at[j]$ := false;
44:     **if** $(state = eating)$ **then** $state := hungry$
45:     **for each** $k$ s.t. $\neg higher[k]$ **do**:
46:         send *switch* msg to $p_k$;    $higher[k] := true$
47: $LinkDown(j)$ indication arrives:
48:     $S := S \setminus \{j\}$
```

$\xi > t_i + \nu * (n-1)$. Thus, by the same definition for T_i as the time of the first event after last node joins LG_i until p_i exits the doorway, it follows that $T_i \leq t_i + \nu * (n-1)$.

If no nodes in LG_i move, the fork collection in Algorithms 1 and 6 is executed in the same way. Thus, Lemma 8 and Lemma 9 hold for Algorithm 6, provided no nodes move from LG_i in the interval $[t_i, T_i + f_i]$. Note that in the proof of both lemmas, T_i is required to be the time of an event after last node joins LG_i until p_i exits the doorway, but not necessary of the first event (it is defined with respect to the first event to derive an upper bound on the response time of the first algorithm). Thus, both lemmas hold for any time $t \geq T_i$; this is used in the proof of the following theorem. Since doorways are not used, the assumption made by these two lemmas that (a hungry static node) p_i is behind the double doorway in some time interval, is replaced by an assumption that p_i is hungry and static in the same interval.

The following theorem summarizes the properties of the second algorithm. Its proof takes care of nodes which leave LG_i due to a movement by restarting the analysis of the response time from the time of the move.

Theorem 25. *The second algorithm satisfies local mutual exclusion property in a MANET, has failure locality* 2 *and response time* $O(n^2)$.

Proof: The local mutual exclusion property is proved in Lemma 3. The proof carries over, since the handling of forks in Algorithm 6 is identical to that of Algorithm 1.

Assume no node in the 2-neighborhood of p_i fails. If p_i eats by time T_i, we are done. Otherwise, if during some time interval $[t, t + f_i]$, where $t \geq T_i$, no node in LG_i moves, we can apply Lemma 9 to deduce that p_i eats by time $t + f_i$.

We now argue that as long as p_i does not move, such an interval is guaranteed to start by time $T_i + (n-1) * f_i$. There are n nodes in LG_i. No nodes join LG_i after time T_i. Thus, eating by p_i is maximally delayed if each node in LG_i other than p_i leaves LG_i by moving just before f_i time has elapsed since the last node moved (or since T_i). After $(n-1) * f_i$ time, there can be no further movement of nodes in LG_i, since p_i remains the only node in LG_i.

Thus p_i eats by time $T_i + (n-1) * f_i + f_i = t_i + \nu * (n-1) + n * f_i = t_i + O(n^2)$, since f_i is in $O(n)$ (recall that $r_i(i)$ is in $O(n)$). □

When the system is static, applying Lemma 9 implies:

Theorem 26. *When there are no changes in the topology, the second algorithm has response time $O(n)$.*

Proof: Recall from Corollary 6 that if p_i becomes hungry at time t_i, no nodes join LG_i after $(n-1)*\nu$ time units. Thus, if p_i eats by time T_i, we are done. Otherwise, we apply Lemma 9 with $T_i + f_i$ and conclude that p_i starts eating by time $t_i + (n-1)*\nu + (n-1)*(2\tau + 4\nu) + 2\tau + 4\nu$, or within $O(n)$ time units of becoming hungry, provided no node in the 2-neighborhood of p_i fails. $\square$

This improves on the best previously known algorithm for the dining philosophers problem in static systems with optimal failure locality [37], which achieved $O(n^2)$ response time. The improvement is achieved due to the notification mechanism, which prevents a thinking node p_j that has a higher priority than its hungry neighbor p_i from interfering with the progress of p_i when p_j becomes hungry.

Chapter 7

Discussion

This thesis proposes 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. The thesis describes and proves the correctness of two algorithms for the problem; the algorithms tolerate node crashes and exhibit different tradeoffs between failure locality and response time.

Our first algorithm consists of two modules, recoloring and fork collection. We proposed two possibilities for the first module exhibiting a tradeoff between practicality and complexity of implementation. Since recoloring is only required when a node moves (and for initialization), the simple 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. The second algorithm, when run in the mobile setting, matches the properties of the best result with optimal failure locality proposed for the static setting. In a static setting, this algorithm outperforms any previous solution with optimal failure locality.

Substituting other distributed coloring procedures in the recoloring module of the first algorithm may provide other solutions for local mutual exclusion with different properties. For example, in this thesis, we have concentrated on deterministic algorithms, but one may apply a randomized approach, e.g., the algorithm proposed by Kuhn and Wattenhofer [19]. Their algorithm has a common scheme of iterative application of one-round color reduction, similar to one of the implementations proposed for the recoloring module in this thesis. As a result, it can easily substitute the coloring procedure used by the recoloring module, leading to an algorithm for local mutual exclusion with probabilistic properties. As stated in Chapter 2, the resulting solution will achieve slightly better response time only if $\delta \gg \log n$. Since this randomized coloring algorithm uses the algorithm of Linial [22] as a subroutine, it has the same assumption on knowing n and δ by all nodes and requires heavy pre-computation.

It is interesting to investigate the inherent complexity of local mutual exclusion. Linial [22] shows a lower bound of $\Omega(\log^* n)$ on the time required to color a static ring of nodes with 3 colors. Since the solution for the dining philosophers problem can be used for coloring (i.e., a node may choose a new color upon the exit from the critical section), this result implies a lower bound of $\Omega(\log^* n)$ for local mutual exclusion 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 [9] 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.

More interestingly, we would like to investigate the cost of node mobility, in particular, whether the cost incurred by our algorithm for recoloring is necessary. The intuition is that all the nodes with the same color could move to become neighbors, thus invalidating the formerly legal coloring and presumably necessitating a larger response time. This argument needs to be made rigorous in order to derive that an algorithm tolerating nodes movement is necessary more expensive (with regard to the response time) than its static counterpart.

Future research should also consider the assumptions made in our model and investigate whether they are necessary. For example, we assume that each of the nodes is aware of its mobility at the time of link creation. This assumption seems necessary in order to achieve good response time since if priority relationship between two neighbors will not relate to their mobility (e.g., node with higher ID has higher priority), it would be possible to create a scenario where a hungry node will be starved by newly arriving neighbors.

Another direction for the future research is to consider the *self-stabilization* property and its variations [11, 12]. It seems our first algorithm can be made *self-organizing* [12] by running a recoloring module to fix the colors of nodes after every topology change. Designing self-stabilizing versions of our algorithms may allow us to further relax the assumptions made in our system model, e.g., FIFO property of the links.

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

Bibliography

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[2] B. Awerbuch and M. E. Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st Symposium on Foundations of Computer Science (FOCS)*, pages 65–74, 1990.

[3] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proc. 7th IEEE Symposium on Computers and Communications (ISCC)*, pages 539–545, 2002.

[4] J. Bar-Ilan and D. Peleg. Scheduling jobs using common resources. *Inf. Comput.*, 125(1):52–61, 1996.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

[7] Y. Chen and J. L. Welch. Self-stabilizing dynamic mutual exclusion for mobile ad hoc networks. *J. Parallel and Distributed Computing*, 65(9):1072–1089, 2005.

[8] M. Choy and A. K. Singh. Tight lower bounds on failure locality of distributed synchronization. In *Proc. 30th Annual Allerton Conf. Communication, Control, and Computing*, pages 127–136, 1992.

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

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

[11] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. of Theoretical Computer Science*, 4:1–40, 1997.

[12] S. Dolev and N. Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In *Proc. Int. Conference On Principles Of Distributed Systems, (OPODIS)*, pages 230–243, 2006.

[13] M. Duflot, L. Fribourg, and C. Picaronny. Randomized dining philosophers without fairness assumption. In *Proc. 2nd Int. Conference on Theoretical Computer Science (TCS)*, pages 169–180, 2002.

[14] 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.

[15] 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.

[16] P. H. Ha, P. Tsigas, M. Wattenhofer, and R. Wattenhofer. Efficient multi-word locking using randomization. In *Proc. 24th Symposium on Principles of Distributed Computing (PODC)*, pages 249–257, 2005.

[17] O. M. Herescu and C. Palamidessi. On the generalized dining philosophers problem. In *Proc. 20th Symposium on Principles of Distributed Computing (PODC)*, pages 81–89, 2001.

[18] J.-R. Jiang. Prioritized h-out of -k mutual exclusion for mobile ad hoc networks and distributed systems. In *Proc. 4th Int. Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 329–334, 2003.

[19] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *Proc. 25th Symposium on Principles of Distributed Computing (PODC)*, pages 7–15, 2006.

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

[21] D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proc. 12th Symposium on Principles of Programming Languages (POPL)*, pages 133–138, 1981.

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

[23] N. A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proc. 12th Symposium on Theory of Computing (STOC)*, pages 70–81, 1980.

[24] N. Malpani, J. L. Welch, and N. H. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proc. 4th Int. workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM)*, pages 96–103, 2000.

[25] A. Mayer, M. Naor, and L. Stockmeyer. Local computations on static and dynamic graphs. In *Proc. 3rd Israeli Symposium on the Theory of Computing and Systems (ISTCS)*, pages 268–278, 1995.

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

[27] R. Mellier, J.-F. Myoupo, and V. Ravelomanana. A non-token-based-distributed mutual exclusion algorithm for single-hop mobile ad hoc networks. In *Proc. Conference on Mobile and Wireless Communication Networks (MWCN)*, pages 287–298, 2004.

[28] M. Nesterenko and A. Arora. Dining philosophers that tolerate malicious crashes. In *Proc. 22nd Int. Conference on Distributed Computing Systems (ICDCS)*, pages 191–198, 2002.

[29] M. Papatriantafilou and P. Tsigas. On distributed resource handling: Dining, drinking and mobile philosophers. In *Int. Conference on Principles of Distributed Systems (OPODIS)*, pages 293–308, 1997.

[30] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[31] S. M. Pike and P. A. Sivilotti. Dining philosophers with crash locality 1. In *Proc. 24th Int. Conference on Distributed Computing Systems (ICDCS)*, pages 22–29, 2004.

[32] S. M. Pike, Y. Song, and S. Sastry. Wait-free dining under eventual weak exclusion. In *Proc. 9th Int. Conference on Distributed Computing and Networking (ICDCN)*, pages 135–146, 2008.

[33] I. Rhee, A. Warrier, and L. Xu. Randomized dining philosophers to TDMA scheduling in wireless sensor networks. Technical Report TR-2005-20, Department of Computer Science, North Carolina State University, 2005.

[34] I. Rhee and J. L. Welch. On the time complexity of the dining philosophers problem. In *Proc. 1st Int. Conference on Parallel and Distributed Processing Techniques and Application (PDPTA)*, 1995.

[35] P. A. Sivilotti, S. M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proc. 12th IASTED Int. Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529, Nov. 2000.

[36] E. Styer and G. L. Peterson. Improved algorithms for distributed resource allocation. In *Proc. 7th Symposium on Principles of Distributed Computing (PODC)*, pages 105–116, 1988.

[37] Y.-K. Tsay and R. Bagrodia. An algorithm with optimal failure locality for the dining philosophers problem. In *Proc. 8th Int. Workshop on Distributed Algorithms (WDAG)*, pages 296–310, 1994.

[38] J. E. Walter, G. Cao, and M. Mohanty. A k-mutual exclusion algorithm for wireless ad hoc networks. In *ACM Workshop on Principles of Mobile Computing (POMC)*, 2001.

[39] 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.

[40] J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proc. 3rd Int. workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM)*, pages 7–14, 1999.

[41] W. Wu, J. Cao, and M. Raynal. A dual-token-based fault tolerant mutual exclusion algorithm for manets. In *Proc. 3rd Int. Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 572–583, 2007.

[42] W. Wu, J. Cao, and J. Yang. A scalable mutual exclusion algorithm for mobile ad hoc networks. In *Proc. 14th Int. Conference on Computer Communications and Networks (ICCCN)*, pages 159–164, 2005.

אנו מציגים שתי גרסאות של מודול הצביעה: הראשון משתמש באלגוריתם לצביעה חמדנית ואילו השני משתמש באלגוריתם לצביעה מהירה של גרפים שהוצג על ידי לינאל. שתי גרסאות מציגות יחסי גומלין בין מימוש פרקטי יותר לבין מדדי ביצועים טובים יותר. גרסת האלגוריתם הראשון למניעה הדדית לוקלית, המשתמשת במודול לצביעה חמדנית, היא בעלת מקומיות נפילה גרועה של n וזמן תגובה $O((n+\delta^3)\delta)$. יחד עם זאת, גרסה זו היא פשוטה למימוש ולכן עשויה להיות פרקטית בסביבות בהן תזוזות אינן נפוצות. הגרסה השנייה, המשתמשת בצביעה המהירה של לינאל, מניחה ידע של n ו-δ בכל אחד מצמתים ומשיגה מדדי ביצועים טובים בהרבה – מקומיות נפילה של $\max(\log^*n,4)+2$ וזמן תגובה $O((\log^*n+\delta^4)\delta)$. למרות שמקומיות הנפילה של הגרסה השנייה אינה אופטימאלית, היא מתחילה ב-6 וגדלה בצורה מאד איטית עם גודל המערכת; זמן התגובה תלוי באופן חלש מאד ב-n והינו פולינומי ב-δ. עובדות אלה מאפשרות יישום של גרסה זו ברשתות עם מספר רב של צמתים, וכן ברשתות דלילות (רשתות בהן לכל צומת מספר קטן של שכנים).

האלגוריתם השני למניעה הדדית לוקלית משיג מקומיות נפילה של 2 וזמן תגובה $O(n^2)$. תוצאה זו, שמושגת ברשתות אד הוק ניידות, משתווה לתוצאה הטובה ביותר בעלת מקומיות נפילה אופטימאלית, שהושגה עבור רשתות סטטיות. תכונה נוספת חשובה של האלגוריתם היא שזמן התגובה שלו הוא $O(n)$ כאשר מריצים אותו בסביבה סטטית, ובמובן זה, הביצועים שלו עולים על כל התוצאות הקודמות. בנוסף, אלגוריתם זה אינו דורש כי הצמתים יידעו את הערכים של n או δ.

האלגוריתם השני משתמש בטכניקת "היפוך קשתות" (link reversal), לפיה לכל צומת מוצמד ערך של *גובה*, אשר משתנה תוך כדי ריצת האלגוריתם. הגובה משמש צמתים לקביעת עדיפות במהלך התמודדות על הגישה לקטע הקריטי, במקום הצבעים כפי שנעשה באלגוריתם הראשון. הקשתות בגרף המייצג את רשת התקשורת, מכוונות מצומת גבוה יותר לשכנו הנמוך יותר. כאשר צומת יוצא מקטע הקריטי, הוא הופך את הקשתות המצביעות אליו וכך מעלה את הגובה שלו (ומוריד את העדיפות, מה שמאפשר לשכניו להתקדם ולהגיע לקטע הקריטי).

במערכת (המשמש כמאגר לשמירת נתונים שנאספו על ידי צמתים מהשטח) או מחשב המריץ מקרן בחדר ישיבות.

צמתים ברשתות אד הוק ניידות עשויים להימצא בתנאיי סביבה קשים. משך פעולתם נקבע לרוב על ידי משך חיי הסוללה שלהם, אשר לעתים אינה ניתנת להחלפה. כתוצאה מכך, צמתים יכולים ליפול, ולכן חשוב לפתח פתרונות חסינים לנפילות. אחד הקריטריונים למדידת פתרונות לבעיית מניעה הדדית לוקלית הינו *מקומיות הנפילה* (*failure locality*), אשר מודד את גודל סביבת הצומת המושפעת מנפילתו. לדוגמה, כאשר צומת p_i נופל, אנו נעדיף שהתקדמות של צמתים אשר רחוקים יחסית מ-p_i לא תושפע. מקומיות הנפילה מאפשרת לנו לכמת עד כמה תהליך קבלת ההחלטות של אלגוריתם הינו "מבוזר".

הקריטריון השני המשמש אותנו בהערכת פתרונות הוא *זמן התגובה* (*response time*), אשר מודד את הזמן מאז שצומת מבקש להגיע לקטע הקריטי עד אשר הוא נכנס לקטע הקריטי. על מנת לחשב מדד זה, אנו מניחים חסם עליון על הזמן בו צומת נמצא בקטע הקריטי שלו ועל הזמן הדרוש להעביר הודעה אחת מצומת כלשהו לשכנו. חסמים אלה לא ידועים לצמתים ומשמשים לצורך אנליזה בלבד.

למיטב ידיעתנו, לא קיימת עבודה קודמת על בעיית מניעה הדדית לוקלית לרשתות אד הוק ניידות. ברשתות סטטיות בעלות תקשורת חוטית, בעיה זו ידועה כבעיית *הפילוסופים הסועדים* (*dining philosophers*) שהוצגה על ידי Dijkstra ונחקרה בהרחבה בעבר. בין השאר, הוכח כי מקומיות הנפילה של בעיית הפילוסופים הסועדים חסומה באופן הדוק על ידי 2. זמן התגובה הטוב ביותר שהושג באלגוריתם עם מקומיות נפילה אופטימאלית הינו $O(n^2)$, כאשר n הינו מספר הצמתים ברשת. עבור מקומיות נפילה כלשהי, זמן תגובה הטוב ביותר שהושג הינו $O(\delta^2)$, כאשר δ הינה הדרגה המכסימלית של צומת כלשהו ברשת (אלגוריתם זה משיג מקומיות נפילה של 4).

כל האלגוריתמים הקודמים הינם מתוחכמים בשל הקושי למצוא פתרונות יעילים לבעיה, אפילו בסביבה סטטית. תרומתנו הינה להסתכל על הבעיה בסביבה מורכבת עוד יותר, בה צמתים יכולים לזוז. הצלחנו לתכנן שני אלגוריתמים יעילים אשר מדגימים יחסי גומלין בין מקומיות הנפילה וזמן התגובה.

ביתר פירוט, התוצאה הראשונה שלנו הינה אלגוריתם מודולארי למניעה הדדית לוקלית ברשתות אד הוק ניידות, אשר מקצה צבעים לצמתים ומשתמש בצבעים אלה לפתרון קונפליקטים בין שכנים שרוצים להגיע לקטע הקריטי בו-זמנית. אנו משתמשים בטכניקה של "מסדרון" (doorway) שהוצגה על ידי Lamport ושימשה במספר עבודות אחרות על מנת להבטיח התקדמות לוקלית של אלגוריתם. האלגוריתם מתגבר על הקשיים שעולים עקב תזוזת צמתים על ידי בחירה מדוקדקת של קבוצות הצמתים אשר מחליפים הודעות ביניהם בעקבות התזוזה של אחד הצמתים בסביבה. עקב תזוזות הצמתים, צמתים מסוימים מריצים מודול לצביעה ובוחרים צבע חדש לעצמם.

תקציר

התקדמות בטכנולוגית תקשורת אלחוטית בשנים אחרונות אפשרה יישום ופריסה של רשתות אד הוק ניידות (mobile ad hoc networks, או בקיצור MANETs). רשתות אלה מורכבות מאוסף יחידות מחשוב ניידות (צמתים) אשר מתקשרות ביניהן בעזרת שידורים אלחוטיים, בדרך כלל ללא תמיכה של תשתית קבועה (כגון, נתבים, תחנות ממסר וכו'). אפליקציות נפוצות של רשתות אלה כוללות פריסה בעתות חירום, במבצעים צבאיים ולצורכי איסוף מידע על הסביבה. למרות שהרבה מהאתגרים בתחום החומרה נפתרו, כתיבת תוכנה עבור רשתות אד הוק ניידות עדיין מהווה אתגר משמעותי. אחד הקשיים הגורמים לכך הוא העובדה שרשת התקשורת משתנה בצורה לא צפויה כאשר צמתים זזים. קיומם של פתרונות בדוקים לבעיות נפוצות ברשתות אד הוק ניידות, שיהוו אבני-בניין בבניית אפליקציות, אמורה לסייע בפיתוח תוכנה בסביבה זו.

מניעה הדדית לוקלית הינה אבן-בניין אשר, לדעתנו, יכולה לסייע באפליקציות שונות. נניח סיטואציה בה לכל צומת ישנו חלק מסוים בקוד, הנקרא *הקוד הקריטי* שלו, שצריך להתבצע לעתים, והביצוע חייב להיות במניעה הדדית ביחס לשכנים איתם הצומת יכול לתקשר ישירות (לצורך העניין, צמתים שנמצאים פיזית קרוב יחסית אל הצומת). אנו מעוניינים באלגוריתם שמבטיח כי שני צמתים שכנים לא יימצאו בקטעים הקריטיים שלהם בו-זמנית. יתר על כן, האלגוריתם צריך להבטיח, תחת תנאים סבירים מסוימים, שאם צומת רוצה להגיע לקטע הקריטי שלו, בסופו של דבר יצליח לעשות כך.

במניעה הדדית (גלובלית), שני צמתים בכל הרשת, ללא תלות במרחק ביניהם, לא יכולים להימצא בקטעים הקריטיים שלהם בו-זמנית. למרות שקיימים בספרות פתרונות לא מעטים לבעיית המניעה ההדדית הגלובלית ברשתות אד הוק ניידות, נראה כי לבעיה זו יש פחות אפליקציות פוטנציאליות מאשר למניעה הדדית לוקלית שנחקרה בעבודה זו.

אלגוריתם למניעה הדדית לוקלית יכול לשמש לשיפור תקשורת ברשות אד הוק ניידות. לדוגמה, בעזרת אלגוריתם כזה צמתים הנמצאים קרוב אחד לשני יכולים להתחרות על גישה אקסקלוסיבית לערוץ תקשורת אלחוטי יעודי למשדר קרקעי או למשדר אווירי, כגון משדר של לווין. אלגוריתם כזה יבטיח כי כל צומת יקבל בסופו של דבר את האפשרות להשתמש בערוץ תקשורת ללא הפרעה מצמתים אחרים בסביבתו. אפליקציה אחרת למניעה הדדית לוקלית הינה לבחור צומת שיקבל גישה לחומרה מיוחדת באיזור בו נמצא הצומת, כגון מחשב חזק יותר

המחקר נעשה בהנחיית פרופ׳ חגית עטיה בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

מניעה הדדית לוקלית יעילה וחסינה
ברשתות אד הוק ניידות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אלכס קוגן

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אייר, תשס״ח חיפה יוני, 2008

מניעה הדדית לוקלית יעילה וחסינה
ברשתות אד הוק ניידות

אלכס קוגן