A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks

Jennifer Walter[1] Jennifer L. Welch[2] Nitin Vaidya[3]

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: {jennyw, welch, vaidya}@cs.tamu.edu

Abstract

A fault-tolerant distributed mutual exclusion algorithm which adjusts to node mobility is presented, along with proof of correctness and preliminary simulation results. The algorithm requires nodes to communicate with only their current neighbors, making it well-suited to the ad hoc environment.

1 Introduction

A mobile *ad hoc* network is a network wherein a pair of nodes communicates by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes. Direct communication is possible only between pairs of nodes that lie within one another's transmission radius. Wireless link "failures" occur when previously communicating nodes move such that they are no longer within transmission range of each other. Likewise, wireless link "formation" occurs when nodes that were too far separated to communicate move such that they are within transmission range of each other. Characteristics which distinguish ad hoc networks from existing distributed networks include frequent and unpredictable topology changes and highly variable message delays. These characteristics make ad hoc networks challenging environments in which to implement distributed algorithms.

Past work on modifying existing distributed algorithms for ad hoc networks includes numerous *routing* protocols (e.g., [12, 8, 5, 17, 10, 14, 18, 6]), *wireless channel allocation* algorithms (e.g., [11]), and protocols for *broadcasting* and *multicasting* (e.g., [5, 9, 16]). *Dynamic networks* are fixed wired networks which share some characteristics of ad hoc networks, since failure and repair of nodes and links is unpredictable in both cases. Research on dynamic networks has focused on *total ordering* [13], *end-to-end communication*, and *routing* (e.g., [1, 2]).

Existing distributed algorithms will run correctly on top of ad hoc routing protocols, since these protocols are designed to hide the dynamic nature of the network topology from higher layers in the protocol stack (see figure 1(a)). Routing algorithms on ad hoc networks provide the ability to send messages from any node to any other node. However, our contention is that efficiency can be gained by developing a core set of distributed algorithms, or primitives, which are aware of the underlying mobility in the network, as shown in figure 1(b). In this paper, we present a *mobility aware* distributed mutual exclusion algorithm to illustrate the layering approach in figure 1(b).

The mutual exclusion problem involves a group of processes, each of which intermittently requires access to a resource or a piece of code called the *critical section* (CS). At most one process may be in the CS at any given time. Providing shared access to resources through mutual exclusion is a fundamental problem in computer science, and is worth considering for the ad hoc environment, where stripped-down mobile nodes may need to share resources.

Distributed mutual exclusion algorithms which rely on the maintenance of a logical structure to provide order and efficiency (e.g., [15, 19]) may be inefficient when run in a mobile environment, where the topology

[1]Supported by GE Faculty of the Future and GAANN fellowships.
[2]Supported in part by NSF PYI grant CCR-9396098.
[3]Supported in part by Texas Advanced Technology Program grant 010115-248 and NSF grant CDA-9529442.

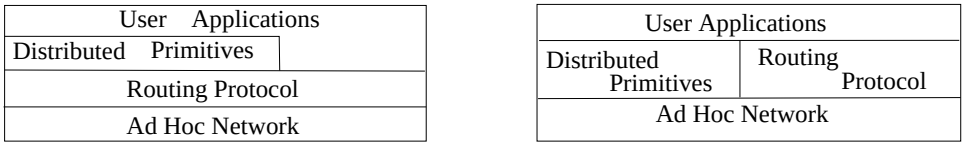

Figure 1: Two possible approaches for implementing distributed primitives

can potentially change with every node movement. Badrinath et al.[3] solve this problem on cellular mobile networks, where the bulk of the computation can be run on wired portions of the network. We present a mutual exclusion algorithm which induces a logical directed acyclic graph (DAG) on the network, dynamically modifying the logical structure to adapt to the changing physical topology in the ad hoc environment.

The next section discusses related work. In section 3, we describe our system assumptions and define the problem in more detail. Section 4 presents our mutual exclusion algorithm. We sketch a proof of correctness and discuss the simulation results in sections 5 and 6, respectively. Section 7 presents our conclusions.

2 Related Work

Token based mutual exclusion algorithms provide access to the CS through the maintenance of a single token which cannot simultaneously be present at more than one node in the system. Requests for CS entry are typically directed to whichever node is the current token holder.

Raymond [19] introduced a token based mutual exclusion algorithm in which requests are sent, over a static spanning tree of the network, toward the token holder; this algorithm is resilient to non-adjacent node crashes and recoveries, but is not resilient to link failures. Chang et al.[4] extend Raymond's algorithm by imposing a logical direction on a sufficient number of edges to induce a *token oriented DAG* in which, for every node i, there exists a directed path originating at i and terminating at the token holder. Allowing request messages to be sent over all edges of the DAG provides resilience to link and site failures. However, this algorithm does not consider link recovery, an essential feature in a system of mobile nodes.

Dhamdhere and Kulkarni [7] show that the algorithm of [4] can suffer from deadlock and solve this problem by assigning a dynamically changing sequence number to each node, forming a total ordering of nodes in the system. The token holder always has the highest sequence number, and, by defining links to point from a node with lower to higher sequence number, a token oriented DAG is maintained. Due to link failures, a node i that wants to send a request for the token may find itself with no outgoing links to the token holder. In this situation, i floods the network with messages to build a temporary spanning tree. Once the token holder becomes part of such a spanning tree, the token is passed directly to node i along the tree, bypassing other requests. Since priority is given to nodes that lose a path to the token holder, it seems likely that other requesting nodes could be starved as long as link failures continue. Also, flooding in response to link failures and storing messages for delivery after link recovery make this algorithm ill-suited to the highly dynamic ad hoc environment.

Our token based algorithm combines ideas from several papers. The partial reversal technique from [10], used to maintain a *destination oriented* DAG in a packet radio network when the destination is static, is used in our algorithm to maintain a token oriented DAG with a dynamic destination. Like the algorithms of [19], [4], and [7], each node in our algorithm maintains a request queue containing the identifiers of neighboring nodes from which it has received requests for the token. Like [7], our algorithm totally orders nodes. The lowest node is always the current token holder, making it a "sink" toward which all requests are

sent. Our algorithm also includes some new features. Each node dynamically chooses its lowest neighbor as its preferred link to the token holder. Nodes sense link changes to immediate neighbors and reroute requests based on the status of the previous preferred link to the token holder and the current contents of the local request queue. All requests reaching the token holder are treated symmetrically, so that requests are continually serviced while the DAG is being re-oriented and blocked requests are being rerouted.

3 System Assumptions

The system contains a set of n independent mobile nodes, communicating by message passing over a wireless network. Assumptions on the mobile nodes and network are:

1. the nodes have unique node identifiers,
2. node failures do not occur,
3. communication links are bidirectional and FIFO,
4. a link-level protocol ensures that each node is aware of the set of nodes with which it can currently directly communicate by providing indications of link formations and failures,
5. incipient link failures are detectable, providing reliable communication on a per-hop basis,
6. message delays obey the triangle inequality (i.e., messages that travel 1 hop will be received before messages sent at the same time that travel more than 1 hop), and
7. partitions of the network do not occur.

In section 7, we discuss the relaxation of some of these assumptions.

Each node is assumed to be running an application whose states are partitioned into three "sections": WAITING (where the node has requested access to the CS), CRITICAL (in which the node is executing the CS), or REMAINDER (where the node is neither requesting nor executing the CS). We assume that 1) nodes cycle through REMAINDER to WAITING to CRITICAL status, in that order and, 2) nodes never stop executing while CRITICAL.

The mutual exclusion problem is to design an algorithm that will interact with the application on each node to ensure

Mutual exclusion: At most one node is in the CS at a given time.

No starvation: Once link failures cease, if a node is waiting to enter the CS, then at a later time that node enters the CS. The hypothesis that link failures cease is needed because an adversarial pattern of link failures can cause starvation.

4 Reverse Link (RL) Mutual Exclusion Algorithm

In this section we first present the data structures maintained at each node in the system, followed by an overview of the algorithm, the algorithm pseudocode, and examples of algorithm operation. Throughout this section, data structures are described for node i, $0 \leq i \leq n-1$. Subscripts on data structures to indicate the node are only included when needed.

4.1 Data Structures

- *status*: Indicates whether node is in the WAITING, CRITICAL, or REMAINDER section. Initially, *status* = REMAINDER.
- N: The set of all nodes in direct wireless contact with node i. Initially, N contains all of node i's neighbors.
- *myHeight*: A three-tuple $(h1, h2, i)$ representing the height of node i. Links are considered to be directed from nodes with higher height toward nodes with lower height, based on lexicographic ordering. E.g., if $myHeight_1 = (2, 3, 1)$ and $myHeight_2 = (2, 2, 2)$, then $myHeight_1 > myHeight_2$ and the link between these nodes would be directed from node 1 to node 2. Initially at node 0, $myHeight_0 = (0, 0, 0)$ and, for all $i \neq 0$, $myHeight_i$ is initialized so that the directed links form a DAG in which every node has a directed path to node 0.
- *height*$[j]$: An array of tuples representing node i's view of $myHeight_j$ for all $j \in N_i$. Initially, *height*$[j] = myHeight_j$, for all $j \in N_i$. From node i's viewpoint, if $j \in N$, then the link between i and j is *incoming* to node i if *height*$[j] >$ *myHeight*, and *outgoing* from node i if *height*$[j] <$ *myHeight*.
- *tokenHolder*: Flag set to true if node holds token and set to false otherwise. Initially, *tokenHolder* = true if $i = 0$, and *tokenHolder* = false otherwise.
- *next*: Indicates the location of the token in relation to i. When node i holds the token, *next* $= i$, otherwise *next* is the node on an outgoing link. Initially, *next* $= 0$ if $i = 0$, and *next* is an outgoing neighbor otherwise.
- Q: Queue which contains identifiers of requesting neighbors. Operations on Q include Enqueue(), which enqueues an item only if it is not already on Q, Dequeue() with the usual FIFO semantics, and Delete(), which removes a specified item from Q, regardless of its location. Initially, $Q = \emptyset$.

4.2 Overview of the RL Algorithm

The mutual exclusion algorithm is event-driven. An event at a node i consists of receiving a message from another node $j \neq i$, or an indication of link failure or formation from the link layer, or an input from the application on node i to request or release the CS. Modules are assumed to be executed atomically. First, we describe the pseudocode triggered by events and then we describe the pseudocode for procedures.

Requesting and releasing the CS: When node i requests access to the CS, it enqueues its own identifier on Q and sets *status* to WAITING. If node i does not currently hold the token and i has a single element on its queue, it calls *ForwardRequest*() to send a *Request* message. If node i does hold the token, i can set *status* to CRITICAL and enter the CS, since it will be at the head of Q. When node i releases the CS, it calls *GiveTokenToNext*() to send a *Token* message if Q is non-empty, and sets *status* to REMAINDER.

Request messages: When a *Request* message sent by a neighboring node j is received at node i, i enqueues j on Q if the link between i and j is incoming at i. If Q is non-empty, and *status* = REMAINDER, i calls *GiveTokenToNext*() provided i holds the token. Non-token holding node i calls *ForwardRequest*() if $|Q| = 1$ or if Q is non-empty and the link to *next* has reversed.

Token messages: When node i receives a *Token* message from some neighbor j, i sets $tokenHolder =$ true. Then i lowers its height to be lower than that of the last token holder, node j, informs all its neighbors of its new height by sending *LinkInfo* messages, and calls *GiveTokenToNext*().

LinkInfo messages: When a *LinkInfo* message is received at node i from node j, j is added to N and j's height is recorded in $height[j]$. If j is an element of Q and j is an outgoing link, then j is deleted from Q. If node i has no outgoing links and is not the token holder, i calls *RaiseHeight*() so that an outgoing link will be formed. If node i is not the token holder, Q is non-empty, and the link to *next* has reversed, i calls *ForwardRequest*() since it must send another *Request* for the token.

Link failures: When node i senses the failure of a link to a neighboring node j, it removes j from N and, if j is an element of Q, deletes j from Q. Then, if i is not the token holder and i has no outgoing links, i calls *RaiseHeight*(). If node i is not the token holder, Q is non-empty, and the link to *next* has failed, i calls *ForwardRequest*() since it must send another *Request* for the token.

Link formation: When node i detects a new link to node j, i sends a *LinkInfo* message to j with *myHeight*.

Procedure ForwardRequest: Selects node i's lowest height neighbor to be *next*. Sends a *Request* message to *next*.

Procedure GiveTokenToNext: Node i dequeues the first node on Q and sets *next* equal to this value. If $next = i$, i enters the CS. If $next \neq i$, i lowers $height[next]$ to $(-\infty, -\infty, next)$, so any incoming *Request* messages will be sent to *next*, sets $tokenHolder =$ false, and then sends a *Token* message to *next*. If Q is non-empty after sending a *Token* message to *next*, a *Request* message is sent to *next* immediately following the *Token* message so the token will eventually be returned to i.

Procedure RaiseHeight: Called at non-token holding node i when i loses its last outgoing link. Node i raises its height (in lines 1-3) using the *partial reversal* method of [10] and informs all its neighbors of its height change with *LinkInfo* messages. All nodes on Q to which links are now outgoing are deleted from Q. If Q is not empty at this point, *ForwardRequest*() is called since i must send another *Request* for the token.

4.3 The RL Algorithm

```
When node i requests access to the CS:
    1. status := WAITING
    2. Enqueue(Q, i)
    3. If (not tokenHolder) then
    4.     If (|Q| = 1) then ForwardRequest()
    5. Else GiveTokenToNext()

When node i releases the CS:
    1. If (|Q| > 0) then GiveTokenToNext()
    2. status := REMAINDER
```

```
When Request(h) received at node i from node j:  // h denotes j's height when message was sent
    1. height[j] := h  // set i's view of j's height
    2. If (myHeight < height[j]) then Enqueue(Q, j)
    3. If (tokenHolder) then
    4.     If ((status = REMAINDER) and (|Q| > 0)) then GiveTokenToNext()
    5. Else  // not tokenHolder
    6.     If ((Q = [j]) or ((|Q| > 0) and (myHeight < height[next]))) then ForwardRequest()

When Token(h) received at node i from node j:  // h denotes j's height when message was sent
    1. tokenHolder := true
    2. height[j] := h
    3. myHeight.h1 := h.h1
    4. myHeight.h2 := h.h2 - 1   // lower my height
    5. Send LinkInfo(myHeight) to all k ∈ N
    6. If (|Q| > 0) then GiveTokenToNext()

When LinkInfo(h) received at node i from node j:  // h denotes j's height when message was sent
    1. N := N ∪ {j}
    2. height[j] := h
    3. If (myHeight > height[j]) then Delete(Q, j)
    4. If ((myHeight < height[k], for all k ∈ N) and (not tokenHolder)) then RaiseHeight()
    5. Else if ((|Q| > 0) and (myHeight < height[next])) then ForwardRequest()  // reroute request

When failure of link to j detected at node i:
    1. N := N − {j}
    2. Delete(Q, j)
    3. If ((myHeight < height[k], for all k ∈ N) and (not tokenHolder)) then RaiseHeight()
    4. Else if ((|Q| > 0) and (next ∉ N)) then ForwardRequest()  // reroute request

When formation of link to j detected at node i:
    1. Send LinkInfo(myHeight) to j

Procedure ForwardRequest():
    1. next := l ∈ N : height[l] ≤ height[j] for all j ∈ N
    2. Send Request(myHeight) to next

Procedure GiveTokenToNext():  // only called when |Q| > 0
    1. next := Dequeue(Q)
    2. If (next ≠ i) then
    3.     tokenHolder := false
    4.     height[next] := (−∞, −∞, next)
    5.     Send Token(myHeight) to next
    6.     If (|Q| > 0) then Send Request(myHeight) to next
    7. Else  // next = i
    8.     status := CRITICAL
    9.     Enter CS

Procedure RaiseHeight():
    1. myHeight.h1 := 1 + min_{k∈N}{height[k].h1}
    2. S := {l ∈ N : height[l].h1 = myHeight.h1}
    3. If (S ≠ ∅) then myHeight.h2 := min_{l∈S}{height[l].h2} − 1
    4. Send LinkInfo(myHeight) to all k ∈ N
       // Raising own height can cause some links to become outgoing
    5. For (all k ∈ N such that myHeight > height[k]) do Delete(Q, k)
       // Must reroute request if queue non-empty, since just had no outgoing links
    6. If (|Q| > 0) then ForwardRequest()
```

4.4 Examples of Algorithm Operation

We first discuss the case of a static network, followed by a dynamic network. An illustration of the algorithm on a static network (in which links do not fail or form) is depicted in figure 2. Snapshots of the state of the system during algorithm execution are shown, with time increasing from 2(a) to 2(e). The direct wireless links are shown as dashed lines connecting circular nodes. The arrow on each wireless link points from the higher height node to the lower height node. The request queue at each node is depicted as a rectangle, the height is shown as a 3-tuple, and the token holder as a shaded circle. The *next* pointers are shown as solid arrows. Note that when a node holds the token, its *next* pointer is directed towards itself.

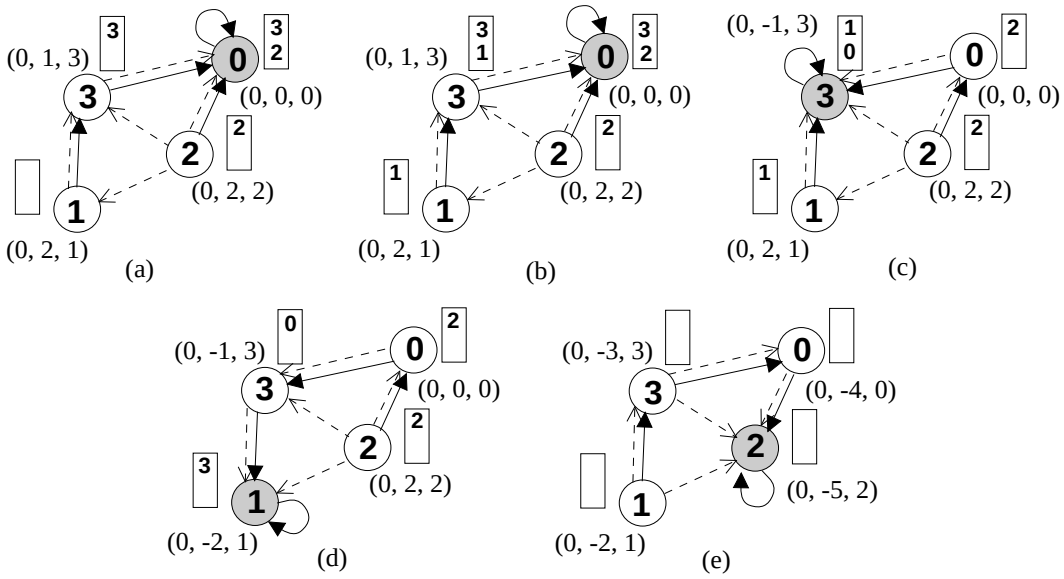

Figure 2: Operation of Reverse Link Mutual Exclusion Algorithm on Static Network

In figure 2(a), nodes 2 and 3 have requested access to the CS (note that nodes 2 and 3 have enqueued themselves on Q_2 and Q_3) and have sent *Request* messages to node 0, which enqueued them on Q_0 in the order in which the *Request* messages were received. Part (b) depicts the system at a later time, where node 1 has requested access to the CS, and has sent a *Request* message to node 3 (note that 1 is enqueued on Q_1 and Q_3). Figure 2(c) shows the system state after node 0 has released the CS and has sent a *Token* message to node 3, followed by a *Request* sent by node 0 on behalf of node 2. Observe that the logical direction of the link between node 0 and node 3 changes from being outgoing at node 3 in part (b), to being incoming at 3 in part (c), when node 3 receives the *Token* message and lowers its height. Notice also the *next* pointers of nodes 0 and 3 change from both nodes having *next* pointers directed toward node 0 in part (b) to both nodes having *next* pointers directed toward node 3 in part (c). Part (d) shows the system state after node 3 sent a *Token* message to node 1, followed by a *Request* message. The *Request* message was sent because node 3 received the *Request* message from node 0. Notice that the items at the head of the nodes' request queues in part (d) form a path from the token holder, node 1, to the sole remaining requester, node 2. Part (e) depicts the state of the system after *Token* messages have been passed from node 1 to 3, node 3 to 0, and from node 0 to 2. Observe that the middle element, $h2$, of each node's *myHeight* tuple decreases by 1 for every hop the token travels, so that the token holder is always the lowest height node in the system.

We now consider the execution of the RL algorithm on a dynamic network. The height information

allows each node i to keep track of the current logical direction of links to neighboring nodes, particularly to the node chosen to be *next*. If the link to *next* changes and $|Q| > 0$, node i must reroute its request by calling *ForwardRequest*().

Figure 3(a) shows the same snapshot of the system execution as is shown in figure 2(a), with time increasing from 3(a) to 3(e). Figure 3(b) depicts the system state after node 3 has moved in relation to the other nodes in the system, resulting in a network which is temporarily not token oriented, since node 3 has no outgoing links. Node 0 has adapted to the lost link to node 3 by removing 3 from its request queue. Node 2 takes no action as a result of the loss of its link to node 3, since the link to $next_2$ was not affected and node 2 still has one outgoing link. In part (c), node 3 has adapted to the loss of its link to node 0 by raising its height and has sent a *Request* message to node 1 (which has not yet arrived at node 1). Part (d) shows the state of the system after node 1 has received the *Request* message from node 3, has enqueued 3 on Q_1, and has raised its height due to the loss of its last outgoing link. In part (e), node 1 has propagated the *Request* received from node 3 by sending a *Request* to node 2, also informing node 2 of the change in its height. Node 2 subsequently enqueued 1 on Q_2, but did not raise its own height or send a *Request*, because node 2 has an intact link to $next_2$, node 0, to which it already sent an unfulfilled request.

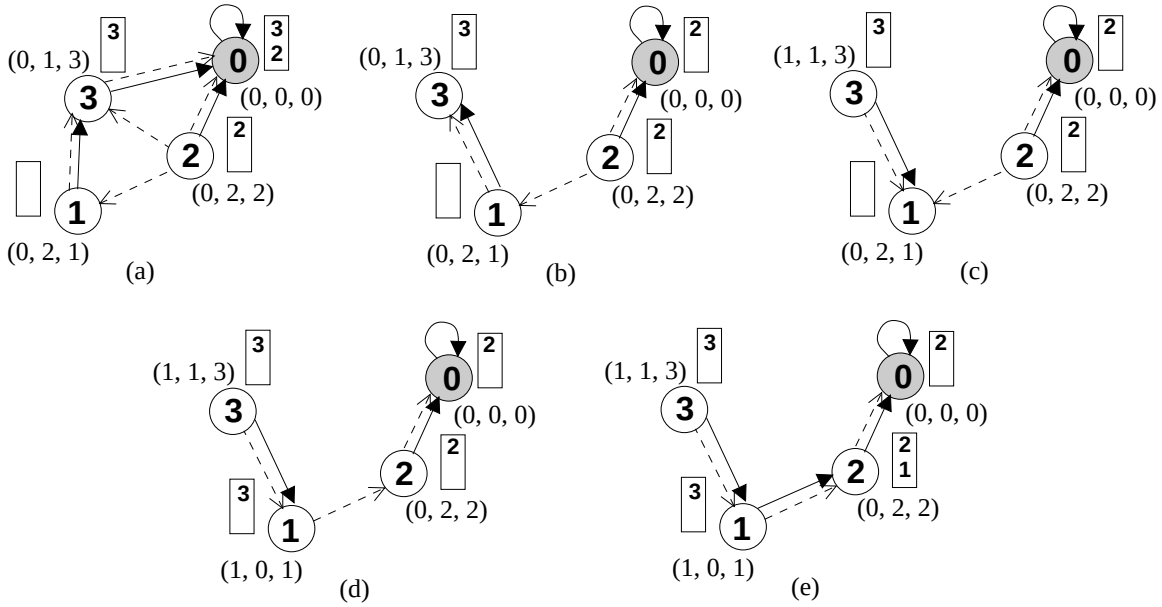

Figure 3: Operation of Reverse Link Mutual Exclusion Algorithm on Dynamic Network

5 Correctness of Reverse Link Algorithm

The following theorem holds because there is only one token in the system at any time.

Theorem 1 *The algorithm ensures mutual exclusion.*

To prove no starvation, we first show that, after link failures cease, eventually the network is in a "good" state, and then we apply a variant function argument.

We will show that after link failures cease, the logical directions on the links imparted by height values will eventually form a "token oriented" DAG. Since the height values of the nodes are totally ordered,

there cannot be any cycles in the logical graph, and thus it is a DAG. The hard part is showing that this DAG is token oriented, defined next.

Definition 1 *A node i is the* token holder *in a state if* $\text{tokenHolder}_i = true$ *or if a* Token *message is in transit from node i to $next_i$.*

Definition 2 *The DAG is* token oriented *in a state if for every node $i, i \in \{0, \ldots, n-1\}$, there exists a directed path originating at node i and terminating at the token holder.*

To prove that the DAG is eventually token oriented, we first show, in Lemma 1, that this condition is equivalent to the absence of "sink" nodes [10], as defined below. We then show, in Lemma 3, that eventually there are no sinks, if link failures cease.

Definition 3 *A node i is a* sink *in a state if*
$(tokenHolder_i = false)$ *and* $((myHeight_i < height_i[j])$, *for all* $j \in N_i)$.

Lemma 1 *In every state of every execution, the DAG is token oriented iff there are no sinks.*

Proof: The only-if direction follows from the definition of a token oriented DAG. The if direction is proved by contradiction. Assume in contradiction that there exists a node i in a state such that $tokenHolder_i =$ false and for which there is no directed path starting at i and ending at the token holder. Since there are no sinks, i must have at least one outgoing link which is incoming at some other node. Since 1) the number of nodes is finite, 2) the network is connected, and 3) all edges are logically directed such that no logical path can form a cycle, there must exist a directed path from i to the token holder, a contradiction. □

To show that eventually there are no sinks, we first show that nodes that hold the token after link failures cease never become sinks subsequently. Let T be this set of nodes. Then we show by induction on d that nodes at distance d from T eventually cease being sinks.

Claim 1 *In every execution, every height increase is caused by either a link failure or the receipt of a* LinkInfo *message sent as a result of a* RaiseHeight *at a neighboring node.*

Proof: By the code, a *RaiseHeight* is caused only by the receipt of a *LinkInfo* message or a link failure. *LinkInfo* messages are sent 1) when a link forms, 2) when a node receives a *Token* message, or 3) when a node does *RaiseHeight*.

Case 1, the receipt of a *LinkInfo* message sent due to link formation, will not cause a node to raise its height. To see this, consider the formation of a new link between two nodes, j and k. This link will be incoming at j and outgoing at k, or vice versa, due to the total ordering on the nodes. Since there are no partitions in the network, node j had at least one link to some other node $p \neq k$ and node k had at least one link to some other node $q \neq j$ prior to the formation of the new link between j and k. Since neither node lost its last outgoing link as a consequence of the formation of the new link, the link formation would cause neither j nor k to raise its height.

We now show that a *LinkInfo* message sent in case 2 will not cause the receiving node to call *RaiseHeight*. Suppose, in contradiction, some node l calls *RaiseHeight* due to the receipt of a *LinkInfo* message sent when a neighboring node k received a *Token* message. Then, at node l, $height_l[k] < myHeight_l$ immediately before the *LinkInfo* message arrives from k. But immediately after the *LinkInfo* message arrives from k, $height_l[k] > myHeight_l$ at l. Therefore, node l must be lower in height when the *LinkInfo* message from k is received than node k was when k sent the *LinkInfo* message. Since a node only lowers its height upon

receiving a *Token* message, and since a token holding node will not raise its height, node l must receive a *Token* message and send a *Token* message before the *LinkInfo* message arrives from k. Since $height_l[k] < myHeight_l$ when the *LinkInfo* message from k is received at l, node l must have sent a *Token* message to node k before receiving the *LinkInfo* message from k. If node l sent this *Token* message to some node $p \neq k$, then $height_l[k] > myHeight_l$ before l received the *LinkInfo* message from k and this receipt would not cause a *RaiseHeight* at l. Because node k sent the *LinkInfo* message when it received the token, this means that after node k sent the *LinkInfo* message, either 1) at least one other node $j \neq l$ received a *Token* message prior to node l receiving the *Token* message, or 2) a *Token* message was sent from node k to node l, arrived at l, and was sent back to k before the *LinkInfo* message from k arrived at l. The triangle inequality assumption on message delay is violated in the first case and the FIFO assumption on message delivery is violated in the second case.

□

As a consequence of claim 1, after link failures cease, a non-token holding node l will lose its last outgoing link only because a neighboring node raised its height.

Claim 2 *In every execution, after link failures cease, any edge over which a* Token *message is sent will not change its direction unless a* Token *message is sent over it in the opposite direction.*

Proof: Consider the first time this claim is violated after link failures cease, and suppose the link between nodes i and j reverses direction after node i sends a *Token* message to node j. Node j must have raised its height due to the loss of its last outgoing link. By the code, we know that node j was not the token holder at the time it raised its height. This means that node j sent a *Token* message to some node $k \neq i$ prior to raising its height. However, by assumption, the reversal of the link between i and j was the first time the claim was violated. Therefore, the link between node j and k must be directed toward k. In this case, node j would not have raised its height, a contradiction.

□

Lemma 2 follows from claims 1 and 2.

Lemma 2 *Every node l which becomes the token holder after link failures cease will always have a directed path to the token holder, and therefore l will never again become a sink.*

Let T be the set of nodes that ever hold the token after link failures cease.

Lemma 3 *Every node l which is not in T is only a sink finitely often.*

Proof: By induction on d, the distance of a shortest path from l to any node in T, among all paths from l to any node in T, in the *undirected* underlying graph.

Basis: $d = 1$. Then l is a neighbor of some node k in T. When k is the token holder, the edge between l and k is directed from l to k. Since k's height never increases, by Lemma 2, and since l's height never decreases subsequently, the edge is always directed from l to k after k is the token holder. Thus l never becomes a sink again.

Inductive step: Suppose Lemma 3 is true for $d - 1$. Let k be a neighbor of l whose distance to T is $d - 1$. Suppose, in contradiction, that l becomes a sink infinitely often. Then l calls *RaiseHeight* infinitely often. Since l never lowers its height, l's height is eventually greater than k's final height (after k does its last *RaiseHeight*). But subsequently the edge between l and k is always directed from l to k, so l would

never be a sink again.

□

Lemma 4 follows from Lemmas 1, 2, and 3.

Lemma 4 *Once link failures cease, the logical direction on links imparted by height values will eventually form a token oriented DAG.*

Consider a node that is WAITING in an execution at some point after link failures have ceased. We first define the "request chain" of a node to be the path along which its request has propagated. Then we modify the variant function argument in [19] to show that the node eventually gets to enter the CS.

Definition 4 *Given a state, a* request chain *for any node l with a non-empty request queue is the maximal length list of node identifiers $p_1 = l, p_2, \ldots, p_j$, where for each i, $1 < i \leq j$,*
p_i's queue is not empty,
$p_i = next_{p_{i-1}}$,
$next_{p_{i-1}}$ is outgoing at p_{i-1} and incoming at p_i,
no Request *message is in transit from p_{i-1} to p_i, and*
no Token *message is in transit from p_i to p_{i-1}.*

Lemma 5, proved by induction on the execution, gives useful information about what is going on at the end of a request chain:

Lemma 5 *The following is true in every state: Let l be a node with a non-empty request queue and let $p_1 = l, p_2, \ldots, p_j$ be l's request chain. Then*

(a) l is in Q_l if l is WAITING,

(b) p_{i-1} is in $Q_{p_i}, 1 < i \leq j$, and

(c) either p_j is the token holder,
or a Token *message is in transit to p_j,*
or a Request *message is in transit from p_j to $next_{p_j}$,*
or a LinkInfo *message is in transit from $next_{p_j}$ to p_j with $next_{p_j}$ higher than p_j,*
or $next_{p_j}$ sees the link to p_j as failed.

Lemma 6 *Once link failures cease, for every state in which a node l's request chain does not include the token holder, then there is a later state in which l's request chain does include the token holder.*

Proof: By Lemma 4, after link failures cease, eventually a token oriented DAG will be formed. Consider a state after link failures cease in which the DAG is token oriented, meaning that all *LinkInfo* messages generated when nodes raise their heights have been delivered.

The proof is by contradiction. Assume node l's request chain never includes the token holder (and that the token is not in transit to a node in l's request chain). So the token can only be held by or be in transit to nodes that are not in l's request chain. By our assumption on the execution, no *LinkInfo* messages caused by a *RaiseHeight* will be in transit to a node in l's request chain, nor will any node in l's request chain detect a failed link to a neighboring node. Therefore, by Lemma 5(c), a *Request* message must be in transit from a node in l's request chain to a node that is not in l's request chain, and the number of nodes in l's request chain will increase when the *Request* message is received. At this point, l's request chain will either include the token holder, another *Request* message will be in transit from a node in l's request chain

to a node that is not in l's request chain, or l's request chain will have joined the request chain of some other node. While the the number of nodes in l's request chain increases, the number of nodes not in l's request chain decreases, since there are a finite number of nodes in the system. So eventually l's request chain includes all nodes. Therefore, if the token is not eventually contained in l's request chain, it is not in the system, a contradiction.

□

Let l be a node that is WAITING after link failures cease. Given a state s in the execution, a function V_l for l is defined to be the following vector of positive integers. Let $p_1 = l, p_2, \ldots, p_m$ be l's request chain. $V_l(s)$ has either $m+1$ or m elements $\langle v_1, v_2, \ldots \rangle$, depending on whether a *Request* message is in transit from p_m or not. In either case, v_1 is the position of $p_1 (= l)$ in Q_l, and for $1 < j \leq m$, v_j is the position of p_{j-1} in Q_{p_j}. (Positions are numbered in ascending order with 1 being the head of the queue.) If present, $v_{m+1} = n+1$. These vectors are compared lexicographically.

Lemma 7 *V_l is a variant function.*

Proof: The key points to prove are:

(1) V_l never has more than n entries and every entry is between 1 and $n+1$, so the range of V_l is well-founded.

(2) Most events can be easily seen not to increase V_l. Here we discuss the remaining events.

When the *Request* message at the end of l's request chain is received by node j from node p_m, V_l decreases from $\langle v_1, \ldots, v_m, n+1 \rangle$ to $\langle v_1, \ldots, v_m, v_{m+1}, \ldots \rangle$, where $v_{m+1} < n+1$ since v_{m+1} is p_m's position in Q_j.

When a *Token* message is received by the node p_m at the end of l's request chain, it is either

- kept at p_m, so V_l decreases from $\langle v_1, \ldots, v_{m-1}, v_m \rangle$ to $\langle v_1, \ldots, v_{m-1}, v_m - 1 \rangle$,
- or sent toward l, so V_l decreases from $\langle v_1, \ldots, v_{m-1}, v_m \rangle$ to $\langle v_1, \ldots, v_{m-1} \rangle$,
- or sent away from l, followed by a *Request* message, so V_l decreases from $\langle v_1, \ldots, v_{m-1}, v_m \rangle$ to $\langle v_1, \ldots, v_{m-1}, v_m - 1, n+1 \rangle$.

(3) To see that the events that cause V_l to decrease will continue to occur, consider the following two cases:

Case 1: The token holder is not in l's request chain. By Lemma 6, eventually the token holder will be in l's request chain.

Case 2: The token holder is in l's request chain. Since no node stays in the critical section forever, at some later time the token will be sent and received, decreasing the value of V_l, by part (2) of this proof.

□

Once V_l equals $\langle 1 \rangle$, l enters the CS. We have:

Theorem 2 *If link failures cease, then every request is eventually satisfied.*

6 Simulation Results

We ran simulations of the RL algorithm on systems of 30 nodes which were initially totally connected. Requests for CS execution were assumed to arrive at a node according to a Poisson distribution with rate λ_1. Since we want to simulate the algorithm with mobility while maintaining a connected graph, link failures are scheduled initially according to a Poisson distribution with rate λ_2. If the link failure will cause a partition, the failure is rescheduled at a later time using the same rate, λ_2. Link recoveries are assumed to occur according to a Poisson distribution with rate λ_3. The time to execute the CS and message propagation delay between any two nodes are both assumed to be one time unit. Two performance measures are considered in this study: 1) the number of messages received per entry to the CS, and 2) the average amount of time a processor waits to gain access to the CS.

Figure 4(a) plots the cost in terms of number of messages received per entry to the CS against values of λ_1 increasing from 10^{-5} requests per time unit to 1 request per time unit from left to right along the x axis for two different rates, λ_2 and λ_3. To obtain an average, we made 5 runs for each value of λ_1 with both combinations of mobility rates. For curve 1 in figure 4(a), $\lambda_2 = .002$ per time unit and $\lambda_3 = .008$ per time unit, so at any given time during execution, approximately 20% of the links are up and 80% of the links are down. In curve 2 in figure 4(a), the values of λ_2 and λ_3 are reversed. Notice that both the x and y axes of figure 4(a) have logarithmic scales, and that many more messages are sent per CS entry when the load on the system is low. With a higher link failure rate and lower load, as seen in curve 1 of figure 4(a), messages are sent to maintain the DAG at all times, even though CS entries are only occasional. At higher values of λ_1, each node frequently has a request pending. Therefore, when a node sends a *Request*, it is likely that its neighbor *next* already has at least its own identifier on its request queue, so requests are not propagated over lengthy paths to the token holder. The performance of our simulation under heavy demand is similar to the results obtained through simulation by [19] and [4] for more static situations.

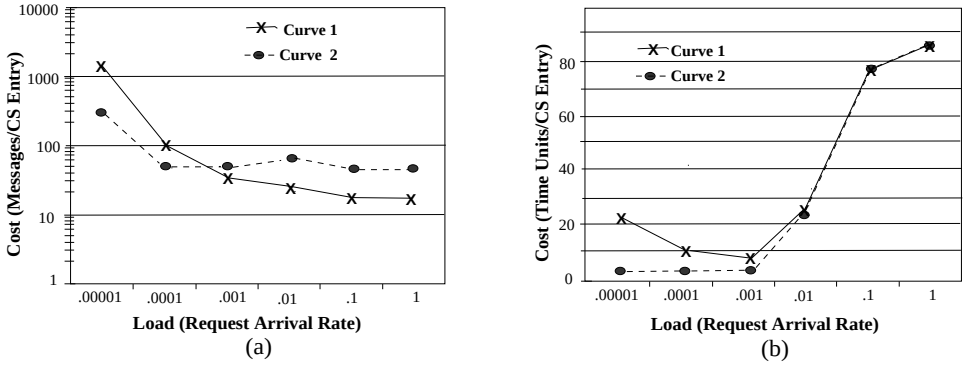

Figure 4: Performance of RL Algorithm

In figure 4(b), the average wait time per CS entry, on the y axis, is plotted against the same request arrival and mobility rates as were used in figure 4(a). From figure 4(b), we can see that the wait time per CS entry increases as the request arrival rate increases, except initially on curve 1, where low connectivity causes greater cost at very low loads. However, at high loads, the wait time appears to be independent of the failure rate (in figure 4(b), curves 1 and 2 overlap for values of λ_1 greater than .01).

These simulation results are preliminary and our interpretation may change as we refine the mobility model to more closely reflect actual mobility patterns. A noticeable omission in the results is a plot of the performance of the RL algorithm against a "static" distributed mutual exclusion algorithm running on top of an ad hoc routing protocol. We believe that the savings in message passing using the RL algorithm could be substantial.

7 Conclusion and Future Work

We presented a distributed mutual exclusion algorithm designed to be aware of and adapt to node mobility, along with a proof of correctness, and preliminary results of simulation, assuming no partitions. We have made small changes to the algorithm so that it solves the mutual exclusion problem when partitions occur, provided the requesting node is in the same connected component as the token holding node for a sufficiently long period. We are experimenting with relaxing the triangle inequality assumption on message delays, a relaxation which we believe can be accomplished through a minor change in the algorithm.

Our work on simulating the algorithm will continue, with future experiments testing the complexity of the RL mutual exclusion algorithm against similar algorithms run on top of an ad hoc routing protocol. One of our major goals is to incorporate realistic mobility models into the simulations, since the assumption of random link failures and formations is not representative of realistic ad hoc mobility patterns. Future simulations will include some parameter which takes into account the likelihood of message collisions in the ad hoc environment as well as the limitations on the transmission capabilities of individual nodes. Longer range future work includes the development and simulation of other mobility aware distributed primitives for use in the ad hoc environment.

Acknowledgements

We thank Savita Kini for many discussions on previous versions of the algorithm and Soma Chaudhuri for careful reading and helpful comments on the liveness proof.

References

[1] Y. Afek, E. Gafni, and A. Rosen. The slide mechanism with applications in dynamic networks. In *Proc. of 11th Annual Symp. on Prin. of Dist. Computing*, pages 35–46, 1992.

[2] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial end to end communication. In *Proc. of 30th Annual Symp. on Found. of Comp. Sci.*, pages 358–363, 1989.

[3] B. R. Badrinath, A. Acharya, and T. Imielinski. Structuring distributed algorithms for mobile hosts. In *Proc. of 14th IEEE Intl. Conf. on Distributed Computing*, pages 21–28, 1994.

[4] Y. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. of 9th IEEE Symp. on Reliable Dist. Systems*, pages 146–154, 1990.

[5] C. Chiang and M. Gerla. Routing and multicast in multihop, mobile wireless networks. In *Proc. of ICUPC '97*, pages 546–551, 1997.

[6] M. S. Corson and A. Ephremides. A distributed routing algorithm for mobile wireless networks. *ACM J. Wireless Networks*, 1(1):61–81, 1997.

[7] D. M. Dhamdhere and S. S. Kulkarni. A token based k-resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50:151–157, 1994.

[8] R. Dube, C. D. Rais, K. Wang, and S. K. Tripathi. Signal stability based adaptive routing (SSA) for ad-hoc mobile networks. *IEEE Personal Communications*, pages 36–45, Feb. 1997.

[9] A. Ephremides and T. V. Truong. Scheduling broadcasts in multihop radio networks. *IEEE Trans. on Communications*, 38(4):456–460, 1990.

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

[11] M. Gerla and T.-C. Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, pages 255–265, 1995.

[12] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In T. Imielinski and H. Korth, editors, *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[13] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proc. of 15th Annual Symp. on Prin. of Dist. Computing*, pages 68–76, 1996.

[14] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. A cluster-based approach for routing in dynamic networks. *Proc. of ACM SIGCOMM Computer Communication Review*, pages 372–378, 1997.

[15] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *Proc of Internat. Conf. on Dist. Comp. Systems*, pages 354–360, 1991.

[16] E. Pagani and G. P. Rossi. Reliable broadcast in mobile multihop packet networks. In *Proc. of ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '97)*, pages 34–42, 1997.

[17] V. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proc. of INFOCOM '97*, pages 1405–1413, 1997.

[18] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing for mobile computers. In *Proc. of ACM SIGCOMM Symp. on Communication, Architectures and Protocols*, pages 234–244, 1994.

[19] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.