A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks *

Jennifer Walter [a,**] Jennifer L. Welch [b,***] Nitin Vaidya [c,****]

[a] *Department of Computer Science, Texas A&M University, College Station, TX 77843-3112*
E-mail: jennyw@cs.tamu.edu
[b] *Department of Computer Science, Texas A&M University, College Station, TX 77843-3112*
E-mail: welch@cs.tamu.edu
[c] *Department of Computer Science, Texas A&M University, College Station, TX 77843-3112*
E-mail: vaidya@cs.tamu.edu

A fault-tolerant distributed mutual exclusion algorithm which adjusts to node mobility is presented, along with proof of correctness and simulation results. The algorithm requires nodes to communicate with only their current neighbors, making it well-suited to the ad hoc environment. Experimental results indicate that adaptation to mobility can improve performance over that of similar non-adaptive algorithms when nodes are mobile.

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

* This is an extended version of the paper presented at the Dial M for Mobility workshop, Dallas TX, Oct. 30, 1998.
** Supported by GE Faculty of the Future and Dept. of Education GAANN fellowships.
*** Supported in part by NSF PYI grant CCR-9396098.
**** Supported in part by Texas Advanced Technology Program grant 010115-248 and NSF grant CDA-9529442.

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., [8,9,11,13,16,18,19,23–25]), *wireless channel allocation* algorithms (e.g., [14]), and protocols for *broadcasting* and *multicasting* (e.g., [8,12,22,28]). *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* [17], *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., [21,26]) may be inefficient when run in a mobile environment, where the topology 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. We then present simulation results comparing the performance of

Figure 1. Two possible approaches for implementing distributed primitives

this algorithm to a static distributed mutual exclusion algorithm running on top of an ad hoc routing protocol. Simulation results indicate that our algorithm has better average waiting time per CS entry and message complexity per CS entry no greater than the cost incurred by a static mutual exclusion algorithm running on top of an ad hoc routing algorithm.

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 [26] 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.[7] 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 [10] show that the algorithm of [7] 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 [13], 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 [26], [7], and [10], 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 [10], 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. partitions of the network do not occur, and
7. messages are delivered in *causal order*[27] [1]

We only explicitly model the mutual exclusion process at each node. Constraints on the behavior of the application processes and the network appear as conditions on executions. The system architecture is shown in Figure 2.

Each node has a (mutual exclusion) *process*, which is modeled as a state machine, with the usual set of states, some of which are initial states, and a transition function. Each state contains a local variable that holds the current neighbors of the node. The transition function is described in more detail shortly.

[1] See Section 7 for a discussion of relaxing assumptions 6 and 7.

Figure 2. System Architecture

A *configuration* describes the instantaneous state of the whole system; formally, it is a set of n states, one for each process on each node. In an *initial* configuration, each state is an initial state.

A *step* of the process at node i is triggered by the occurrence of an input event. Input events are:

- $\mathrm{RequestCS}_i$: the application process on node i requests access to the critical section
- $\mathrm{ReleaseCS}_i$: the application process on node i releases its access to the critical section
- $\mathrm{Recv}_i(j, m)$: node i receives message m from the mutual exclusion process on node j
- $\mathrm{LinkUp}_i(l)$: node i receives notification that the link l incident on i is now up
- $\mathrm{LinkDown}_i(l)$: node i receives notification that the link l incident on i is now down

The effect of a step is to apply the process' transition function, taking as input the current state of the process and the input event, and producing as output an (optional) output event and a new state for the process. Output events are:

- $\mathrm{EnterCS}_i$: inform the application process on node i that it can enter the critical section

- $\text{Send}_i(j, m)$: send message m to node j from node i

The only constraint on the state produced by the transition function is that the neighbor set variable of i must be properly updated in response to a LinkUp or LinkDown event.

RequestCS_i, EnterCS_i, and ReleaseCS_i are called *application events*, while Send_i, Recv_i, LinkUp_i, and LinkDown_i are called *network events*.

An *execution* is a sequence of the form $C_0, in_1, out_1, C_1, in_2, out_2, C_2, \ldots$, where the C_k's are configurations, the in_k's are input events, and the out_k's are sets of output events. An execution ends in a configuration if it is finite. A positive real number is associated with each in_i, representing the time at which that step occurs. An execution must satisfy a number of additional conditions, which we now list. The first set of conditions are basic "syntactic" ones.

- It starts with an initial configuration.
- For each $k \geq 1$, if in_k occurs at node i, then out_k and i's state in C_k are correct according to i's transition function operating on in_k and i's state in C_{k-1}.
- The times assigned to the steps must be nondecreasing. If the execution is infinite, then the times must increase without bound. At most one step by each process can occur at a given time.

The next set of conditions require the application process to interact properly with the mutual exclusion process.

- If in_k is RequestCS_i, then the previous application event at node i (if any) is ReleaseCS_i.
- If in_k is ReleaseCS_i, then the previous application event at node i must be EnterCS_i.

The last set of conditions constrain the behavior of the network to match the informal description given above.

- $\text{LinkUp}_i(l)$ occurs at time t iff $\text{LinkUp}_j(l)$ occurs at time t (where l joins i and j). Furthermore, $\text{LinkUp}_i(l)$ only occurs if j is currently a neighbor of i (according to i's neighbor variable). The analogous condition holds for LinkDown.
- At a given time t, a particular link either goes up or down but not both.
- If $\text{Recv}_i(j, m)$ occurs at some time t, then $\text{Send}_j(i, m)$ occurs at some previous time t' and between t' and t the link between i and j is continuously up.

I.e., the network does not duplicate or corrupt messages nor deliver spurious messages.

- If $\mathrm{Send}_i(j, m)$ occurs at some time t, then $\mathrm{Recv}_j(i, m)$ occurs at some later time t' and between t and t' the link between i and j is continuously up. I.e., messages are not lost. (Note that this imposes some restrictions on when LinkDown events can occur.)
- If $\mathrm{Send}_j(i, m)$ happens before (in the sense of [?]) $\mathrm{Send}_{j'}(i, m')$, then $\mathrm{Recv}_i(j, m)$ precedes $\mathrm{Recv}_i(j', m')$. This condition is a form of causal ordering (e.g., [27]); in particular, it means that if j sends a message m to i, then m is received before any message that is the end of a chain of messages initiated by j after j sent m. It also implies FIFO ordering on each link.
-

Now we can state the problem formally. In every execution, the following must hold:

- If in_k is $\mathrm{EnterCS}_i$, then the previous application event at node i must be $\mathrm{RequestCS}_i$ (so critical section access is only given to requesting nodes).
- Mutual Exclusion: If in_k is $\mathrm{EnterCS}_i$, then any previous $\mathrm{EnterCS}_j$ event must be followed by a $\mathrm{ReleaseCS}_j$ prior to in_k.
- No Lockout: If in_k is $\mathrm{RequestCS}_i$, then there is a following $\mathrm{EnterCS}_i$.

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 $j \in N_i$. In 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 outgoing 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 ($myHeight.h1$, $myHeight.h2-1$, $next$), 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 [13] 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. Send *LinkInfo*($h.h1, h.h2 - 1, i$) to all outgoing $k \in$ N
4. *myHeight.h1* := *h.h1*
5. *myHeight.h2* := *h.h2* - 1 // lower my height
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 := $N \cup \{j\}$
2. $height[j]$:= h
3. If ($myHeight$ > $height[j]$) then $Delete(Q, j)$
4. If (($myHeight$ < $height[k]$, $\forall k \in N$) and ($/tokenHolder$)) then $RaiseHeight()$
5. Else if (($|Q| > 0$) and ($myHeight$ < $height[next]$)) then $ForwardRequest()$

When failure of link to j detected at node i:

1. N := $N - \{j\}$
2. $Delete(Q, j)$
3. If (not $tokenHolder$) then
4.   If ($myHeight$ < $height[k]$, $\forall k \in N$) then $RaiseHeight()$
5.   Else if (($|Q| > 0$) and ($next \notin N$)) then $ForwardRequest()$

When formation of link to j detected at node i:

1. Send $LinkInfo(myHeight)$ to j

Procedure $ForwardRequest()$:

1. $next$:= $l \in N$: $height[l]$ $\leq$ $height[j]$ $\forall j \in$ N
2. Send $Request(myHeight)$ to $next$

Procedure $GiveTokenToNext()$: // only called when $|Q| > 0$

1. $next$:= $Dequeue(Q)$
2. If ($next \neq i$) then
3. $tokenHolder$:= false
4. $height[next]$:= ($myHeight.h1$, $myHeight.h2 - 1$, $next$)
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 + $\min_{k \in N}\{height[k].h1\}$
2. $S := \{l \in N$: $height[l].h1$ = $myHeight.h1\}$
3. If ($S \neq \emptyset$) then $myHeight.h2$:= $\min_{l \in S}\{height[l].h2\}$ $-$ 1
4. Send $LinkInfo(myHeight)$ to all $k \in N$
   // Raising own height can cause some links to become outgoing
5. For (all $k \in N$ 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 3. Snapshots of the state of the system during algorithm execution are shown, with time increasing from 3(a) to 3(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 3. Operation of Reverse Link Mutual Exclusion Algorithm on Static Network

In figure 3(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 3(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 4(a) shows the same snapshot of the system execution as is shown in figure 3(a), with time increasing from 4(a) to 4(e). Figure 4(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

Figure 4. Operation of Reverse Link Mutual Exclusion Algorithm on Dynamic Network

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.

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 $tokenHolder_i = \text{true}$ 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 [13], 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

$$(tokenHolder_i = \text{false}) \text{ and } ((myHeight_i < height_i[j]), \text{ for all } j \in N_i).$$

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 $tokenHolder_i = \text{false}$ 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 some time t_1, k receives the token, sets its height to h, and sends h in a *LinkInfo* message to l. It is always true that the token holder has the smallest height of all the nodes. Thus l's height at time t_1 is larger than h. By assumption, when this message is received by l at time t_3, it causes l to lose its last outgoing link, and thus l's height is smaller than h at time t_3. Therefore at some time t_2 in between t_1 and t_3, node l received the token and lowered its height. Since the links are FIFO, l did not receive the token directly from k, since the token is sent after the *LinkInfo* message. Thus, l received the token indirectly from k, violating the causal ordering of messages.

■

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 returned to the sender.

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 claim 1. 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 claim 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 [26] 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,
- $p_i = \mathrm{next}_{p_{i-1}}$,
- the edge between p_{i-1} and p_i 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

In this section we discuss the static and dynamic performance of the Reverse Link (RL) algorithm compared to a mutual exclusion algorithm designed to operate on a static network. We simulated Raymond's token based mutual exclusion algorithm [26] as if it were running on top of a "routing" layer that always provided shortest path routes between nodes. In this section, we will refer to this simulation as "Raymond's with routing" (RR). Raymond's algorithm was used because it is the static algorithm from which the RL algorithm was adapted and because it does not provide for link failures and recovery and must rely on the routing layer to maintain logical paths if run in a dynamic network. In order to make our results more generally applicable, we made best-case assumptions about the underlying routing protocol used with Raymond's algorithm: that it always provides shortest paths and its time and message complexity are zero. If our simulation shows that the RL algorithm is better than the RR combination in some scenario, then the RL algorithm will also be better than Raymond's algorithm in that scenario when *any* real ad hoc routing algorithm is used. If our simulation shows that the RL algorithm is worse than the RR combination in some scenario, then it might or might not be worse in an actual situation, depending on how much worse it is in the simulation and what are the costs of the routing algorithm.

We simulated a 30 node system under various scenarios. We chose to simulate on a 30 node system because for networks larger than 40 nodes the time needed for simulation was very high. Also, we envision ad hoc networks to be much smaller scale than wired networks like the Internet. Typical numbers of nodes used for simulations of ad hoc networks range from 10 to 50 [4–6,15,18,28]. In all experiments, each CS execution took one time unit and each message delay was one time unit. Requests for the CS were modeled as a Poisson process with arrival rate λ_{req}. Thus the time delay between when a node left the CS and made its next request to enter the CS is an exponential random variable with mean $\frac{1}{\lambda_{req}}$ time units. Link changes were modeled as a Poisson process with arrival rate λ_{mob}. Thus the time delay between each change to the graph is an exponential random variable with mean $\frac{1}{\lambda_{mob}}$ time units. Each change to the graph consisted of the deletion of an edge chosen at random (whose loss did not disconnect the graph) and the formation of an edge chosen at random.

In each execution, we measured the average waiting time for CS entry, that is, the average number of time units that nodes spent in their entry sections. We also measured the average number of messages sent per CS entry.

We varied the load on the system (λ_{req}), the degree of mobility (λ_{mob}), and the "connectivity" of the graph. Connectivity was measured as the percentage of possible edges that were present in the graph. Note that a clique on 30 nodes has 435 (undirected) edges.

In the graphs of our results, each plotted point represents the average of six repetitions of the simulation. Thus in plots of average time per CS entry, each point is the average of the averages from six executions, and similarly for plots of average number of messages per CS entry.

For the RR simulations, we initially formed a random connected graph with the desired number of edges and then used breadth-first search to form a spanning tree of the graph to play the part of the static virtual spanning tree over which nodes communicate in Raymond's algorithm. After the spanning tree was formed, we randomly permuted the graph while maintaining the desired connectivity and then calculated the shortest paths from all nodes to their neighbors in the virtual spanning tree. After this, we started the mutual exclusion algorithm and began counting messages and waiting time per CS entry. When link changes occurred, we did not measure the time or messages needed to recalculate shortest path routes in the modified graph. We did measure any added time and distance that the application messages traveled due to route changes, charging one message per link traversed.

For simulations of RL, we formed a random connected graph with the desired number of edges, initialized the node heights and link directions, and then started the algorithm and performance measurements. When link changes occurred, the time and messages needed to find new routes between nodes were included in the overall cost of performance.

In this section, part (a) of each figure displays results when the graph is static, part (b) when $\lambda_{mob} = 10^{-2}$ (low mobility), and part (c) when $\lambda_{mob} = 10^{-1}$ (high mobility). Our choice for the value of the low mobility parameter corresponds to the situation where nodes remain stationary for a few tens of seconds after moving and prior to making another move. Our choice for the value of the high mobility parameter represents a much more volatile network, where nodes remain static for only a few seconds between moves.

6.1. Average waiting time per CS entry

Figure 5 plots the average number of time units elapsed between host request and subsequent entry to the CS against values of λ_{req} increasing from 10^{-4} (the mean time units between requests is 10^4) to 1 (the mean time units between requests is 1) from left to right along the x axis. We chose the high load value of λ_{req} because at this rate each node would have a request pending almost all

Figure 5. Load vs. Time/CS entry for (a) zero, (b) low, and (c) high mobility

the time. The low load value of λ_{req} represents a much less busy network, with requests rarely pending at all nodes at the same time. Plots are shown for runs with 20% (87 links) and 80% (348 links) connectivity for both the RL and RR simulations.

Figure 5 indicates that RL has better performance than RR in terms of average waiting time per CS entry, up to a factor of six. The reason is that Raymond's algorithm sends application messages over a static virtual spanning tree; when a message is sent from a node to one of its neighbors in the virtual spanning tree, it may actually be routed over a long distance, thus increasing the time delay. In contrast, the RL algorithm uses accurate information about the actual topology, resulting in less delay between each request and subsequent CS entry.

Both algorithms show an increase in average waiting time per CS entry from low to high load in figure 5. The higher the load, the larger is the number of other nodes that precede a given node into the CS.

The average waiting time for each CS entry reaches its peak for the RL simulation at around 60 time units per CS entry under the highest load. This is caused by an essentially round robin pattern of token traversal. However, the average waiting time for the RL simulation in figure 5(c) and figure 6(c) at the highest load actually *decreases* under high mobility. This phenomenon may be due to the fact that, at high loads, frequent link failures break the fair pattern in which the token is received, and causing some nodes to get the token more frequently.

Figure 5 also shows that the waiting time advantage of RL over RR increases with increasing load and increasing mobility. The increased waiting time of RR with increased load when the network connectivity is low is due to longer average route lengths. In the simulation trials, the average route length roughly doubled when the connectivity decreased from 80% to 20%. The performance gap between waiting time for RL and RR is seen to a lesser degree at high connectivity, when average route length in RR is lower. However, it is apparent that the RR simulation suffers from the combined effects of higher contention and imposed static spanning tree communication paths at high loads, while RL is mainly affected by contention for the CS at high loads.

Finally, figure 5 suggests that connectivity in the range tested is immaterial to the behavior of the RL algorithm at high load, whereas a larger connectivity is better for RR than a smaller connectivity at all loads. In order to further study the effect of connectivity, we ran the experiments shown in figure 6: the average number of time units elapsed between host request and subsequent entry to the CS is plotted against network connectivity increasing from 10% (43 links) to 100% (435 links) along the x axis. Curves are plotted for low load, where $\lambda_{req} = 10^{-3}$ (the mean time unit between requests is 10^3) and high load, where $\lambda_{req} = 1$ (1 mean time unit between requests) for both the RL and RR simulations.

Figure 6 confirms that connectivity does not affect the waiting time per CS entry in the RL simulation at high load. At high load, the RL algorithm does not exploit connectivity, particularly in the static case. When load is high, the RL simulation always sends request messages over the path last traveled by the

Figure 6. Connectivity vs. Time/CS entry for (a) zero, (b) low, and (c) high mobility

token, even if there is a shorter path to the token when the request is made. At low load in RL, connectivity does affect the waiting time per CS entry because request messages are not always sent over the path last traveled by the token. This is because with lower load there is sufficient time between requests for token movement to change link direction in the vicinity of the token holder, an effect that increases with higher connectivity, shortening request paths.

The waiting time for the RR algorithm decreases with increasing connectivity, since the path lengths between neighbors in the virtual spanning tree approach one. However, even with a clique, when shortest path lengths are all one, the time for RR does not match that for RL. The reason is that the spanning tree used by RR for all communication might have a relatively large diameter, whereas in RL neighboring nodes are always in direct communication.

The results of the simulations in this section are summarized in Table 1.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
RR high load	185	107	185	140	294	290
RL high load	70	60	70	65	60	60
RR low load	17	8	39	25	60	35
RL low load	8	5	6	5	5	5

Table 1: Summary of time per CS entry.

6.2. Average number of messages per CS entry

The RR algorithm sends request and token messages along the virtual spanning tree. Each message from a node to its virtual neighbor is converted into a sequence of actual messages, which traverse the (current) shortest path from the sender to the recipient.

The RL algorithm sends request and token messages along the actual token-oriented DAG. In addition, as the token traverses a path, each node on that path sends *LinkInfo* messages to all its outgoing neighbors. Additional *LinkInfo* messages are sent, and propagated, when a link failure causes a node to lose its last outgoing link.

Our experimental results reflect the relative number of routing messages for RR vs. *LinkInfo* messages for RL. When interpreting these results, it is important to remember that *the simulation of the RR algorithm is not charged for messages needed to recalculate the routes due to topology changes.* Thus, if RL is better than RR in some situation, it will certainly be better when routing messages are charged to it, even if they are prorated. Also, if RR is better than RL in another situation, depending on how much better it is, RL might be comparable or even better than RR when routing messages are charged to RR.

Figure 7 plots the average number of messages received per CS execution against values of λ_{req} ranging from 10^{-4} (the mean time units between requests is 10^4) to 1 (the mean time units between requests is 1) from left to right along the x axis. Plots are shown for runs with 20% (87 links) and 80% (348 links)

Figure 7. Load vs. Messages/CS Entry for (a) zero, (b) low, and (c) high mobility

connectivity for both the RL and RR simulations.

Figure 7 shows that the RR algorithm sends fewer messages than the RL algorithm in all cases studied, except in the static case with low load.

In all situations studied, except the RL simulation in the static case with high connectivity, the number of messages per CS entry decreases with load. The reason is that, although the overall number of messages increases with load in both algorithms, due to the additional token and request messages, it increases less than linearly with the number of requests, and hence less than linearly with the number of CS entries. In the extreme, at very high load, every time the token moves, it is likely to cause a CS entry.

In the static case with high connectivity, the RL algorithm experiences a threshold effect around load of .01: when load is less than .01, the number of messages per CS entry is roughly constant at a lower value, and when the load is above .01, the number of messages per CS entry is roughly constant at a higher value. The threshold effect dies away with decreasing connectivity. We conjecture that some qualitative behavior of the algorithm on a 30 node graph changes when load increases from .001 to .01. This change may be attributed to the observation that token movement more effectively shortens request path length at high connectivity with low load. This is because at low load there is sufficient time between requests for nodes to receive *LinkInfo* messages sent as the token moves, causing them to send requests over direct links to the token holder rather than over the last link on which they sent the token. This effect is amplified at high connectivity because each node is more likely to be directly connected to the token holder.

The RL algorithm sends more messages per CS entry than the RR algorithm when mobility causes link changes, and the number of messages sent in the RL algorithm grows very large under low loads, as can be observed in figure 7(b) and (c). When links fail and form, the RL algorithm sends many *LinkInfo* messages to maintain the token oriented DAG, resulting in a higher message to CS entry ratio at low loads when the degree of mobility remains constant. However, when interpreting these results, it is important to note that the RL algorithm is being charged for the cost of routing in the simulations with mobility, while the RR simulation is not charged for routing.

Figure 8 shows the results of experiments designed to understand the effect of connectivity on the number of messages per CS entry. In the figure, the average number of messages per CS entry is plotted against network connectivity increasing from 10% (43 links) to 100% (435 links) from left to right on the x axis. Curves are plotted for low load, where $\lambda_{req} = 10^{-3}$ (the mean time units between requests is 10^3) and high load, where $\lambda_{req} = 1$ (the mean time units between requests is 1) for both the RL and RR simulations.

In the static case, the number of RL messages per CS entry increases linearly with connectivity, for a fixed load. As connectivity increases, the number of neighbors per node increases, resulting in more *LinkInfo* messages being sent as the token travels. However, the number of RR messages per CS entry decreases (less than linearly) with connectivity, since the shortest path lengths between neighbors in the virtual spanning tree decrease. In fact, our results for RR at

Figure 8. Connectivity vs. Messages/CS Entry for (a) zero, (b) low, and (c) high mobility

100% connectivity (when the virtual spanning tree is an actual spanning tree) and high load match the performance of approximately 4 messages per CS entry cited by Raymond [26] at high load.

Part (a) of figure 8 shows that in the static case the RL algorithm uses fewer messages per CS entry below 30% connectivity for high load and below 70% connectivity for low load.

Figure 8(b) and (c) show that, in the dynamic cases, the number of messages per CS entry is little affected by connectivity for a fixed load. In the RL algorithm, there are two opposing trends with increasing connectivity that appear to cancel each other out: higher connectivity means more neighbors per node, which means more *LinkInfo* messages will be sent with each failure. On the other hand, more neighbors per node means that it is less likely for a link failure to be that of the last outgoing link, and thus *LinkInfo* messages due to failure will propagate less. For the RR case, the logarithmic scale on the y axis in figure 8(c) hides the slight decrease in messages per CS entry, making both curves appear flat.

The results of the simulations in this section are summarized in Table 2.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
RR high load	13	6	11	7	30	20
RL high load	7	25	20	25	140	140
RR low load	27	13	35	20	60	50
RL low load	8	17	180	180	1600	1500

Table 2: Summary of messages per CS entry.

7. Conclusion and Discussion

We presented a distributed mutual exclusion algorithm designed to be aware of and adapt to node mobility, along with a proof of correctness, and simulation results comparing the performance of this algorithm to that of a static token based mutual exclusion algorithm running on top of an ideal ad hoc routing protocol. We assumed there were no partitions in the network throughout this paper for simplicity; partitions can be handled in our algorithm by using a method similar to that used in the TORA ad hoc routing protocol [23]. In [23], additional labels are used to represent the heights of nodes, allowing nodes to detect, by recognition of the originator of a chain of height increases, when a series of height changes has occurred at all reachable nodes without encountering the "destination". A similar partition detection mechanism could be encorporated into our mutual exclusion algorithm at the expense of slightly larger messages.

The assumption that message delivery obeys causal ordering is needed in order to prove that nodes will eventually stop raising their heights after link changes cease. If this assumption is not made, we have a counter example demonstrating how the receipt of a *LinkInfo* message sent as a result of a token delivery can cause the receiver to raise its height. To address this problem, we have developed a simple addition to our algorithm that makes the causal ordering assumption on messages unnecessary, as long as each link remains FIFO. Briefly, when a node receives a token, it sends a *LinkInfo* message to the node from which it received the token. A node sending the token remembers the last node it sent a token to and the height that node should have upon receiving the token. It also sets a flag to indicate that it has not received a *LinkInfo* message from the node to which it last sent the token. While this flag is set, any *LinkInfo* message coming from the last node the token was sent to is ignored unless it has a height matching the expected height of the next token holder, in which case the flag is reset. The flag is also reset if the link fails before the *LinkInfo* with the expected height is received.

Our algorithm compares favorably to the layered approach using an ad hoc routing protocol, providing better average waiting time per CS entry in all tested scenarios. Our simulation results indicate that in many situations the message complexity per CS entry of our algorithm would not be greater than the message cost incurred by a static mutual exclusion algorithm running on top of an ad hoc routing algorithm, when messages of both the mutual exclusion algorithm and the routing algorithm are counted.

Acknowledgements

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

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] S. Basagni, I. Chlamtac, and V. R. Syrotiuk. A distance routing effect algorithm for mobility (DREAM). In *Proc. of 4th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 76–84, 1998.

[5] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, and J. Jetcheva. A performance comparison

of multi-hop wireless ad hoc network routing protocols. In *Proc. of 4th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 85–97, 1998.

[6] R. Casteñeda and S. R. Das. Query localization techniques for on-demand routing protocols in ad hoc networks. In *Proc. of 5th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 186–194, 1999.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *Proc. of 5th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 195–206, 1999.

[16] 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.

[17] 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.

[18] Y. B. Ko and V. H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Proc. of 4th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 66–75, 1998.

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

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[21] 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.

[22] E. Pagani and G. P. Rossi. Reliable broadcast in mobile multihop packet networks. In

Proc. of 3rd ACM/IEEE International Conference on Mobile Computing and Networking, pages 34–42, 1997.

[23] 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.

[24] 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.

[25] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

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

[27] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[28] E. M. Royer and C. E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *Proc. of 5th ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, pages 207–218, 1999.