A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks [*]

Jennifer E. Walter [a,**] Jennifer L. Welch [a,***] Nitin H. Vaidya [a,****]

[a] *Department of Computer Science, Texas A&M University, College Station, TX 77843-3112*

E-mail: jennyw@cs.tamu.edu, welch@cs.tamu.edu, vaidya@cs.tamu.edu

A fault-tolerant distributed mutual exclusion algorithm that 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 communicate move such that they are within transmission range of each other. Characteristics that 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 environ-

[*] 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 and NSF grant CCR-9972235.

[****] Supported in part by Texas Advanced Technology Program grant 010115-248 and NSF grants CDA-9529442 and CCR-9972235.

ments in which to implement distributed algorithms.

Past work on modifying existing distributed algorithms for ad hoc networks includes numerous *routing* protocols (e.g., [8,9,11,13,16,18,19,22–24]), *wireless channel allocation* algorithms (e.g., [14]), and protocols for *broadcasting* and *multicasting* (e.g., [8,12,21,26]). *Dynamic networks* are fixed wired networks that 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, that 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).

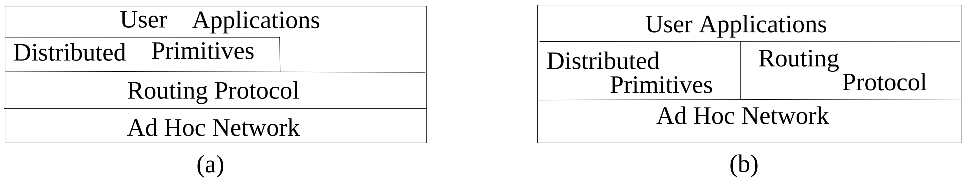

Figure 1. Two possible approaches for implementing distributed primitives

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 that rely on the maintenance of a logical structure to provide order and efficiency (e.g., [20,25]) 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 that 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 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 present 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 that 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 [25] 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 links 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 links 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 [25], [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. Definitions

The system contains a set of n independent mobile nodes, communicating by message passing over a wireless network. Each mobile node runs an application process and a mutual exclusion process that communicate with each other to ensure that the node cycles between its REMAINDER section (not interested in the CS), its WAITING section (waiting for access to the CS), and its CRITICAL section. Assumptions[1] 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,

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

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, and

6. partitions of the network do not occur.

The rest of this section contains our formal definitions. We explicitly model only 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.

We assume the node identifiers are $0, 1, \ldots, n-1$. Each node has a (mutual exclusion) *process*, 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 node identifier and a local variable that holds the current neighbors of the node. The transition function is described in more detail shortly.

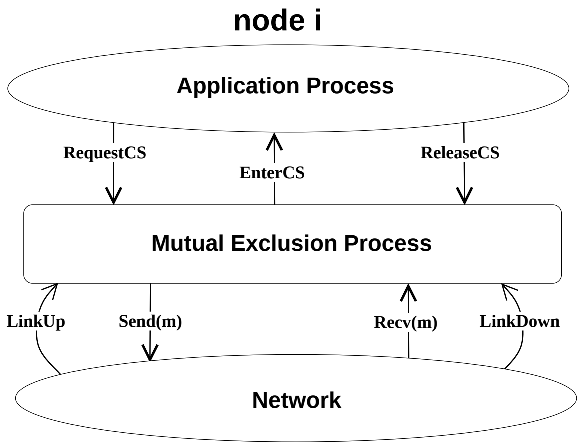

Figure 2. System Architecture

A *configuration* describes the instantaneous state of the whole system; formally, it is a set of n states, one for each process. In an *initial* configuration, each state is an initial state and the neighbor variables describe a connected undirected graph.

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

- RequestCS$_i$: the application process on node i requests access to the CS, entering its WAITING section.
- ReleaseCS$_i$: the application process on node i releases its access to the CS, entering its REMAINDER section.
- Recv$_i(j, m)$: node i receives message m from node j.
- LinkUp$_i(l)$: node i receives notification that the link l incident on i is now up.
- 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 a (possibly empty) set of output events and a new state for the process. *Output events* are:

- EnterCS$_i$: the mutual exclusion process on node i informs its application process that it can enter the CRITICAL section.
- Send$_i(j, m)$: node i sends message m to node j.

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 must end in a configuration if it is finite. A positive real number is associated with each in_i, representing the time at which that event occurs. An execution must satisfy a number of additional conditions, which we now list. The first set of conditions are basic "syntactic" ones.

- C_0 is an initial configuration.
- 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 and to give up the CS in finite time.

- If in_k is $\mathrm{RequestCS}_i$, then the previous application event at node i (if any) is $\mathrm{ReleaseCS}_i$.
- If in_k is $\mathrm{ReleaseCS}_i$, then the previous application event at node i must be $\mathrm{EnterCS}_i$.
- If out_k is $\mathrm{EnterCS}_i$, then there is a following $\mathrm{ReleaseCS}_i$.

The remaining conditions constrain the behavior of the network to match the informal description given above. First, we consider the mobility notification.

- $\mathrm{LinkUp}_i(l)$ occurs at time t if and only if $\mathrm{LinkUp}_j(l)$ occurs at time t, where l joins i and j. Furthermore, $\mathrm{LinkUp}_i(l)$ only occurs if j is currently not a neighbor of i (according to i's neighbor variable). The analogous condition holds for LinkDown.
- A LinkDown never disconnects the graph.

Finally, we consider message delivery. There must exist a one-to-one and onto correspondence between the occurrences of $\mathrm{Send}_j(i, m)$ and $\mathrm{Recv}_i(j, m)$, for all i, j and m. This requirement implies that every message sent is received and the network does not duplicate or corrupt messages nor deliver spurious messages. Furthermore, the correspondence must satisfy the following:

- If $\mathrm{Send}_i(j, m)$ occurs at some time t, then the corresponding $\mathrm{Recv}_j(i, m)$ occurs at some later time t', and the link connecting i and j is continuously up between t and t'. This implies that a LinkDown event for link l cannot occur if any messages are in transit on l.

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

- If out_k is $\mathrm{EnterCS}_i$, then the previous application event at node i must be $\mathrm{RequestCS}_i$. I.e., CS access is only given to requesting nodes.
- *Mutual Exclusion*: If out_k is $\mathrm{EnterCS}_i$, then any previous $\mathrm{EnterCS}_j$ event must be followed by a $\mathrm{ReleaseCS}_j$ prior to out_k.
- *No Starvation*: If there are only a finite number of LinkUp_i and $\mathrm{LinkDown}_i$ events, then if in_k is $\mathrm{RequestCS}_i$, then there is a following $\mathrm{EnterCS}_i$.

For the last condition, the hypothesis that link changes cease is needed because an adversarial pattern of link changes 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 $j \in N_i$. 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*: 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 containing 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, $Q = \emptyset$.
- *receivedLI*[j]: Boolean array indicating whether *LinkInfo* message has been received from node j, to which a *Token* message was recently sent. Any height

information received at node i from a node j for which *receivedLI*$[j]$ is false will not be recorded in *height*$[j]$. Initially, *receivedLI*$_i[j]$ = true for all $j \in N_i$.

- *forming*$[j]$: Boolean array set to true when link to node j has been detected as forming and reset to false when first *LinkInfo* message arrives from node j. Initially, *forming*$_i[j]$ = false for all $j \in N_i$.
- *formHeight*$[j]$: An array of tuples storing value of *myHeight* when new link to j first detected. Initially, *formHeight*$_i[j]$ = *myHeight*$_i$ for all $j \in N_i$.

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. Each message sent includes the current value of *myHeight* at the sender. 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 ignores the *Request* if *receivedLI*$[j]$ is false. Otherwise, i changes *height*$[j]$, and 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 *RaiseHeight*() if the link to j is now incoming and i has no outgoing links or i calls *ForwardRequest*() if $Q = [j]$ 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*(). Node i also informs j of its new height so that j will know that i received the token.

LinkInfo messages: If *receivedLI*[j] is true when a *LinkInfo* message is received at node i from node j, j's height is saved in *height*[j]. If *receivedLI*[j] is false, i checks if the height of j in the message is what it was when i sent the *Token* message to j. If so, i sets *receivedLI*[j] to true. If *forming*[j] is true, the current value of *myHeight* is compared to the value of *myHeight* when the link to j was first detected, *formHeight*[j]. If *myHeight* and *formHeight*[j] are different, then a *LinkInfo* message is sent to j. Identifier j is added to N and *forming*[j] is set to false. 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. Otherwise, if 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, sets *receivedLI*[j] to true, 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*, sets *forming*[j] to true, and sets *formHeight*[j] = *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, sets *receivedLI*[*next*] to 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. If ($receivedLI[j]$) then
 2.    $height[j]$ := $h$  // set i's view of j's height
 3.    If ($myHeight < height[j]$) then $Enqueue(Q, j)$
 4.    If ($tokenHolder$) then
 5.       If (($status$ = REMAINDER) and ($|Q| > 0$)) then $GiveTokenToNext()$
 6.    Else  // not tokenHolder
 7.       If ($myHeight < height[k]$, $\forall\ k \in N$) then $RaiseHeight()$
 8.       Else if (($Q = [j]$) or (($|Q| > 0$) and ($myHeight < height[next]$))) then
 9.          $ForwardRequest()$   // reroute request

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 $k \in N$ and to $j$
 4. $myHeight.h1$ := $h.h1$
 5. $myHeight.h2$ := $h.h2$ - 1   // lower my height
 6. If ($|Q| > 0$) then $GiveTokenToNext()$ Else $next$ := $i$
```

```
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. If (($forming$[$j$]) and ($myHeight \neq formHeight$[$j$])) then
 3.    Send $LinkInfo$($myHeight$) to $j$
 4. $forming$[$j$] := false
 5. If ($receivedLI$[$j$]) then $height$[$j$] := $h$
 6. Else if ($height$[$j$] = $h$) then $receivedLI$[$j$] := true
 7. If ($myHeight$ > $height$[$j$]) then $Delete(Q, j)$
 8. If (($myHeight$ < $height$[$k$], $\forall k \in N$) and (not $tokenHolder$)) then $RaiseHeight$()
    // reroute request
 9. 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. $receivedLI$[$j$] := true
 4. If (not $tokenHolder$) then
 5.    If ($myHeight$ < $height$[$k$], $\forall k \in N$) then $RaiseHeight$()
    // reroute request
 6.    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 $j$
 2. $forming$[$j$] := true
 3. $formHeight$[$j$] := $myHeight$

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.    $receivedLI$[$next$] := false
 6.    Send $Token$($myHeight$) to $next$
 7.    If ($|Q| > 0$) then Send $Request$($myHeight$) to $next$
 8. Else  // next = i
 9.    $status$ := CRITICAL
```

```
10.    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 system configuration 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.

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 configuration 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 directed away from node 3 in part (b), to being directed toward node 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

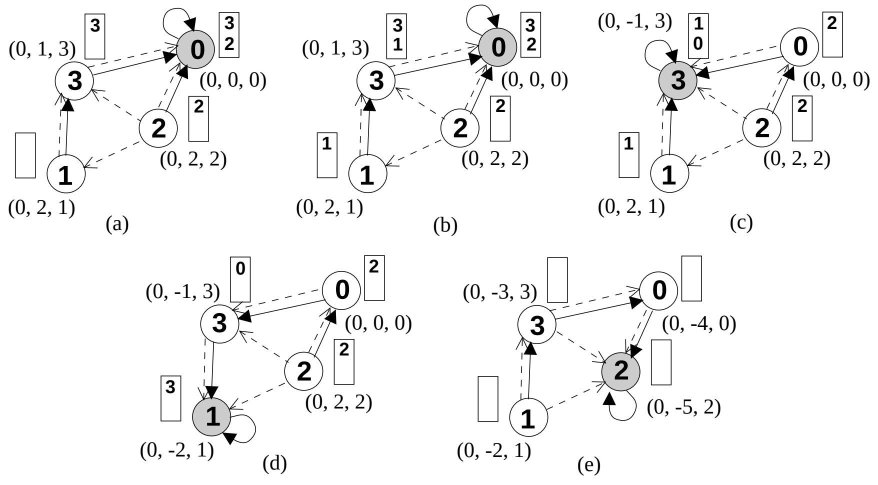

Figure 3. Operation of Reverse Link Mutual Exclusion Algorithm on Static Network

in part (c). Part (d) shows the system configuration 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 system configuration 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 configuration after node 3 has moved in relation to the other nodes in the system, resulting in a network that 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

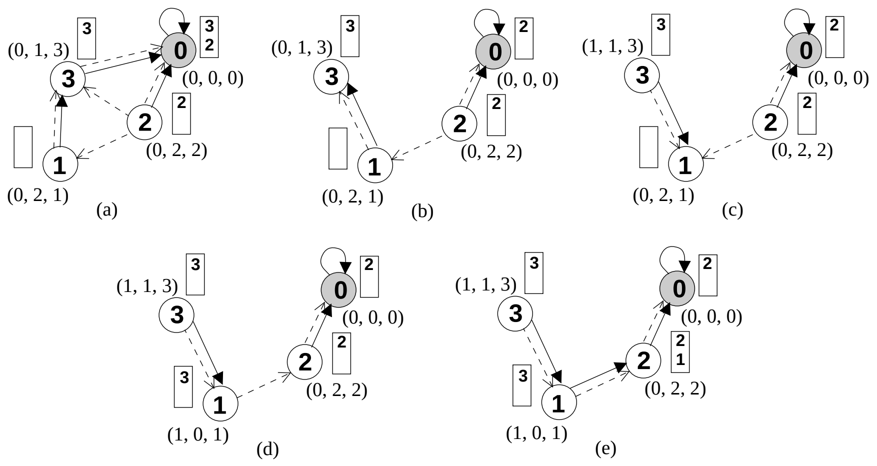

Figure 4. Operation of Reverse Link Mutual Exclusion Algorithm on Dynamic Network

node 0 by raising its height and has sent a *Request* message to node 1 (that has not yet arrived at node 1). Part (d) shows the system configuration 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 changes cease, eventually the system reaches a "good" configuration, and then we apply a variant function argument.

We will show that after link changes 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 configuration if $\mathit{tokenHolder}_i =$ true or if a *Token* message is in transit from node i to next_i.

Definition 2. The DAG is *token oriented* in a configuration 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 Lemma 3, 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 2, that eventually there are no more calls to *RaiseHeight*(). Throughout, we assume that eventually link changes cease.

Definition 3. A node i is a *sink* in a configuration if

$$(\mathit{tokenHolder}_i = \text{false}) \text{ and } ((\mathit{myHeight}_i < \mathit{height}_i[j]), \text{ for all } j \in N_i).$$

Lemma 1. In every configuration of every execution, the DAG is token oriented if and only if 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 configuration such that $\mathit{tokenHolder}_i =$ 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 that is incoming at some other node. Since the number of nodes is finite, the network is connected, and all links 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 (Lemma 3), we show that there are only a finite number of calls to *RaiseHeight*().

Lemma 2. In every execution with a finite number of link changes, there exists a finite number of calls to *RaiseHeight*().

Proof: In contradiction, consider an execution with a finite number of link changes but an infinite number of calls to *RaiseHeight*(). Then, after link changes cease, some node calls *RaiseHeight*() infinitely often. We first note that if one node calls *RaiseHeight*() infinitely often, then every node calls *RaiseHeight*() infinitely often. To see this, consider that a node i would call *RaiseHeight*() infinitely often only if it lost all its outgoing links infinitely often. But this would happen infinitely often at node i only if a neighboring node j raised its height infinitely often, and neighboring node j would only call *RaiseHeight*() infinitely often if its neighbor k raised its height infinitely often, and so on. However, Claim 1 shows that at least one node calls *RaiseHeight*() only a finite number of times.

Claim 1. No node that holds the token after the last link change ever calls *RaiseHeight*() subsequently.

Proof: Suppose the claim is false, and some node that holds the token after the last link change calls *RaiseHeight*() subsequently. Let i be the first node to do so. By the code, node i does not hold the token when it calls *RaiseHeight*(). Suppose that node i sends the token to neighboring node j at time t_1, setting its view of j to be outgoing, and at a later time, t_3, node i calls *RaiseHeight*(). The reason i calls *RaiseHeight*() at time t_3 is that it lost its last outgoing link. Thus, at time t_2 between time t_1 and t_3, the link between i and j has reversed direction in i's view from outgoing to incoming. By the code, the direction change at node i must be due to the receipt of a *LinkInfo* or *Request* message from node j. We discuss these cases separately below.

Case 1: The direction change at node i is due to the receipt of a LinkInfo *message from node j at time t_2.* By the code, when i sends the token to j at t_1, it sets *receivedLI*[j] to false. Therefore, when the *LinkInfo* message is received at i from j at time t_2, node i must have already reset *receivedLI*[j] to true or i would still see the link to j as outgoing and would not call *RaiseHeight*() at time t_2. Since i called *RaiseHeight*() after receiving the *LinkInfo* message from j at time t_2, i must have received the *LinkInfo* message node j sent when it received the token from i *before* time t_2, by the FIFO assumption on message delivery. Then node j must have received the token and sent it to another node, $k \neq i$, after which j raised its height and sent the *LinkInfo* message that node i received at time t_2. However, this violates our assumption that i is the first node to call *RaiseHeight*() after the last link change, a contradiction.

Case 2: The direction change at node i is due to the receipt of a Request *message from node j at time t_2.* By a similar argument to case 1, any *Request* received from node j would be ignored at node i as long as *receivedLI*$[j]$ is false. But this means that node j must have called *RaiseHeight*() after it received the token from node i and subsequently sent the *Request* received by i at time t_2. Again, this violates the assumption that i is the first node to call *RaiseHeight*() after the last link change, a contradiction.

Therefore, node i will not call *RaiseHeight*() at time t_2 and the claim is true.

■

Therefore, by Claim 1, there is only a finite number of calls to *RaiseHeight*() in any execution with a finite number of link changes.

■

Lemma 3 follows from Lemma 2, since if a node becomes a sink, it will eventually be informed via *LinkInfo* messages and will then call *RaiseHeight*().

Lemma 3. Once link changes cease, the logical direction on links imparted by height values will eventually always form a token oriented DAG.

Consider a node that is WAITING in an execution at some point after link changes and calls to *RaiseHeight*() 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 [25] to show that the node eventually gets to enter the CS.

Definition 4. Given a configuration, 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,
- $p_i = next_{p_{i-1}}$,
- the link 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 4 gives useful information about what is going on at the end of a request chain:

Lemma 4. The following is true in every configuration: 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 iff 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.

Proof: By induction on the execution.

Property (a) can easily be shown to hold, since a node enqueues its own identifier when its application requests access to the CS, at which point it changes its status to WAITING. By the code, at no point will a node dequeue its own identifier until just before it enters the CS and sets its status to CRITICAL.

Properties (b) and (c) are vacuously true in the initial configuration, since no node has a non-empty queue.

Suppose (b) and (c) are true in the $(t-1)^{st}$ configuration, C_{t-1}, of the execution. It is possible to show these properties are true in the t^{th} configuration, C_t, by considering in turn every possibility for the t^{th} event. Most of the events applied to C_{t-1} are easily shown to yield a configuration C_t in which properties (b) and (c) are true. Here we discuss the events for which the outcome is less clear by presenting the problematic cases that can appear to disrupt a request chain. We note that, in the following cases, non-token holding nodes are often required to find an outgoing link due to link reversals or failures. It is not hard to show that a node i that is not the token holder can always find an outgoing link due to the performance of *RaiseHeight*().

Case 1: Node i receives a Request*(h) from node j and does not enqueue j on its request queue.* To ensure that j's *Request* is not overlooked, causing possible starvation, we show that either a *LinkInfo* or a *Token* message is sent to j from i if a *Request* from j is received at i and j is not enqueued.

Case 1.1: receivedLI[j] is false at i. It must be that i sent the token to j in some previous configuration and i has not yet received the *LinkInfo* message that j

must send to i upon receipt of the token. If the token is not in transit from i to j or held by j in C_{t-1}, then earlier j had the token and passed it on. The *Request* received by i was sent before the *LinkInfo* message that j must send to i upon receipt of the token. So if j is WAITING in C_{t-1}, it has already sent a newer *Request* and properties (b) and (c) hold for this request chain in C_t by the inductive hypothesis.

Case 1.2: receivedLI[j] is true at i. Then if j is not enqueued on i's request queue, it must be that $myHeight_i > h$. Since j viewed i as outgoing when it sent the *Request*, node i must have either called *RaiseHeight*() after j was in N_i or the relative heights of i and j changed between the time link (i, j) was first detected and before j was added to N_i. In either case, node j must eventually receive a *Linkinfo* message from i and see that its link to $next_j$ has reversed, in which case j will take action resulting in the eventual sending of another *Request.*

Case 2: Node i receives an input causing it to delete identifier j from its request queue. To ensure that j's *Request* is not forgotten when i calls *Delete*(Q, j), we show that either node j received a *Token* message prior to the deletion, in which case j's *Request* is satisfied, or node j is notified that the link to i failed, in which case j will take the appropriate action to reroute the request chain.

Case 2.1: Node i calls *Delete*(Q, j) because it receives a *LinkInfo* message from j indicating that i's link to j has become outgoing at i. Then, since i enqueued j, it must be that in some earlier configuration i saw the link to j as incoming. Since the receipt of the *LinkInfo* message from j caused the link to change from incoming to outgoing in i's view, it must be that the *LinkInfo* was sent by j when j received the token and lowered its height. If the token is not held by j in C_{t-1}, then earlier j had the token and passed it on. If j is WAITING in C_{t-1}, it has already sent a newer *Request* and properties (b) and (c) hold for this request chain in C_t by the inductive hypothesis.

Case 2.2: Node i calls *Delete*(Q, j) because it received an indication that link (i, j) failed. Then j must receive the same indication, in which case it can take appropriate action to advance any request chains.

Case 3: Node i receives an input which makes it see the link to $next_i$ as incoming or failed. In this case, any request chains including node i in C_{t-1} end at i in C_t. We show that node i takes the correct action to propagate these request chains by sending either a new *Request* or a *LinkInfo* message.

Case 3.1: Node i receives a *LinkInfo* message from neighbor $j = next_i$ indicating that i's link to j has become incoming at i. If the link to j was i's last outgoing link, then in C_t i will call *RaiseHeight*(). Node i will delete the identifiers of any nodes on outgoing links from its request queue. Node i will send a *LinkInfo* message to each neighbor, including nodes whose identifiers were removed from i's request queue. If i's request queue is non-empty it will call *ForwardRequest*() and send a *Request* message to the node chosen as $next_i$ in C_t.

Case 3.2: Node i receives an indication that the link to $next_i$ has failed. In C_t, i will take the same actions as it did in case 3.1, when its link to $next_i$ reversed.

Therefore, no action taken by node i can make properties (b) and (c) false and the lemma holds. ■

Lemma 5. Once link changes and calls to *RaiseHeight*() cease, for every configuration in which a node l's request chain does not include the token holder, then there is a later configuration in which l's request chain does include the token holder.

Proof: By Lemma 3, after link changes cease, eventually a token oriented DAG will be formed. Consider a configuration after link changes and calls to *RaiseHeight*() 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. 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 call to *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 4(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 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 changes and calls to *RaiseHeight*() cease. Given a configuration 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 a *Request* message is in transit, then $V_l(s)$ has $m+1$ elements and $v_{m+1} = n+1$; otherwise, $V_l(s)$ has only m elements. These vectors are compared lexicographically.

Lemma 6. 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, l's request chain increases in length to $m+1$, 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 after the *Request* message is received.

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 5, 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 CS 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 changes 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 [25] 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 30 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 [4–6,15,18,26].

In all our 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 a link chosen at random (whose loss did not disconnect the graph) and the formation of a link 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 WAITING 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 links that were present in the graph. Connectivity values presented in this section represent *initial* graph connectivity. Note that a clique on 30 nodes has 435 (undirected) links.

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 links 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 links, 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

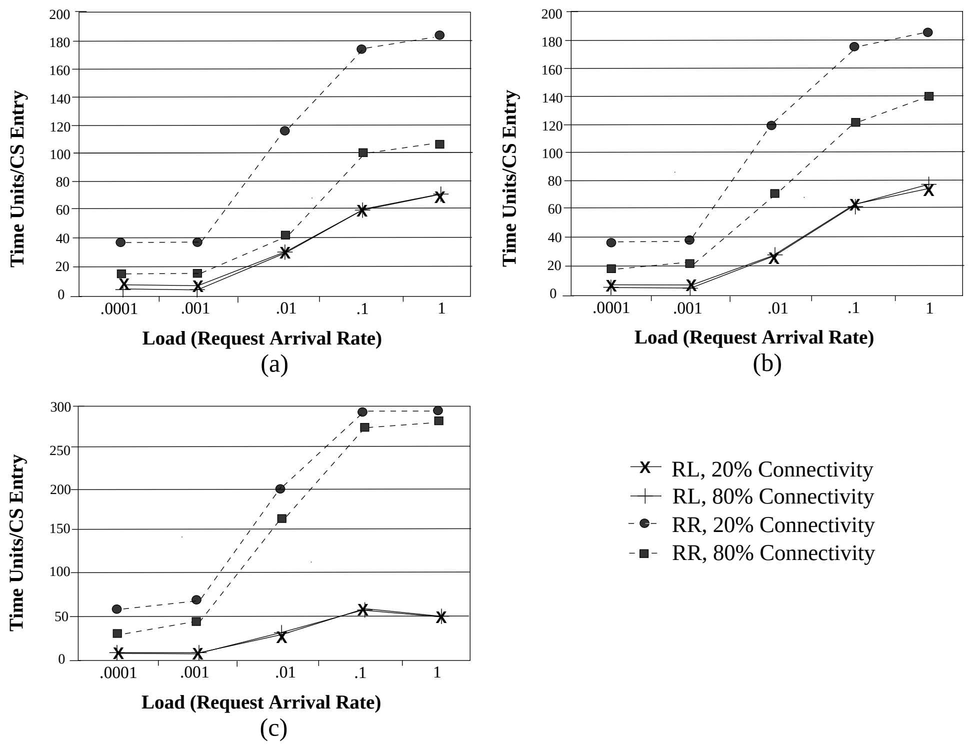

Figure 5. Load vs. Time/CS entry for (a) zero, (b) low, and (c) high mobility

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 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 75 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) 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, 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.

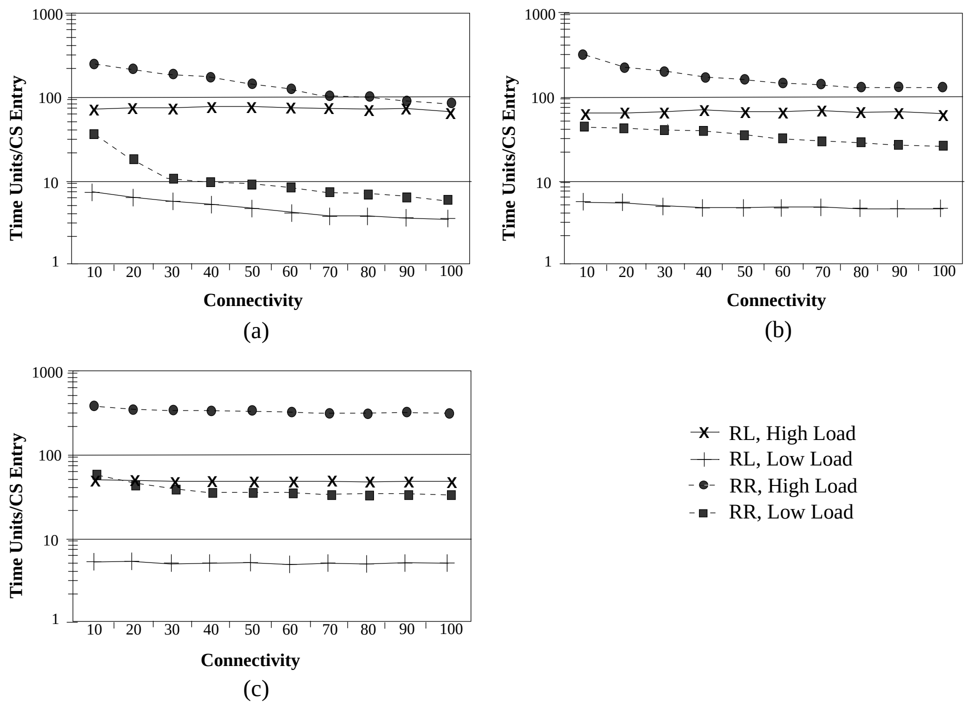

Figure 6. Connectivity vs. Time/CS entry for (a) zero, (b) low, and (c) high mobility

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. When load is high, the RL simulation always sends request messages over the path last traveled by the 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. This table includes data points from both sets of graphs depicted in this subsection. The chosen data points show average waiting time for high (80%) and low (20%) connectivity and for high and low loads in all mobility scenarios.

Table 1
Summary of time per CS entry.

	Zero Mobility		Low Mobility		High Mobility	
	20%[a]	*80%*[a]	*20%*[a]	*80%*[a]	*20%*[a]	*80%*[a]
RR high load	185	107	185	140	294	290
RL high load	75	75	63	63	49	49
RR low load	17	8	39	25	60	35
RL low load	7	4	5	5	6	7

[a] Initial network connectivity.

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, that 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

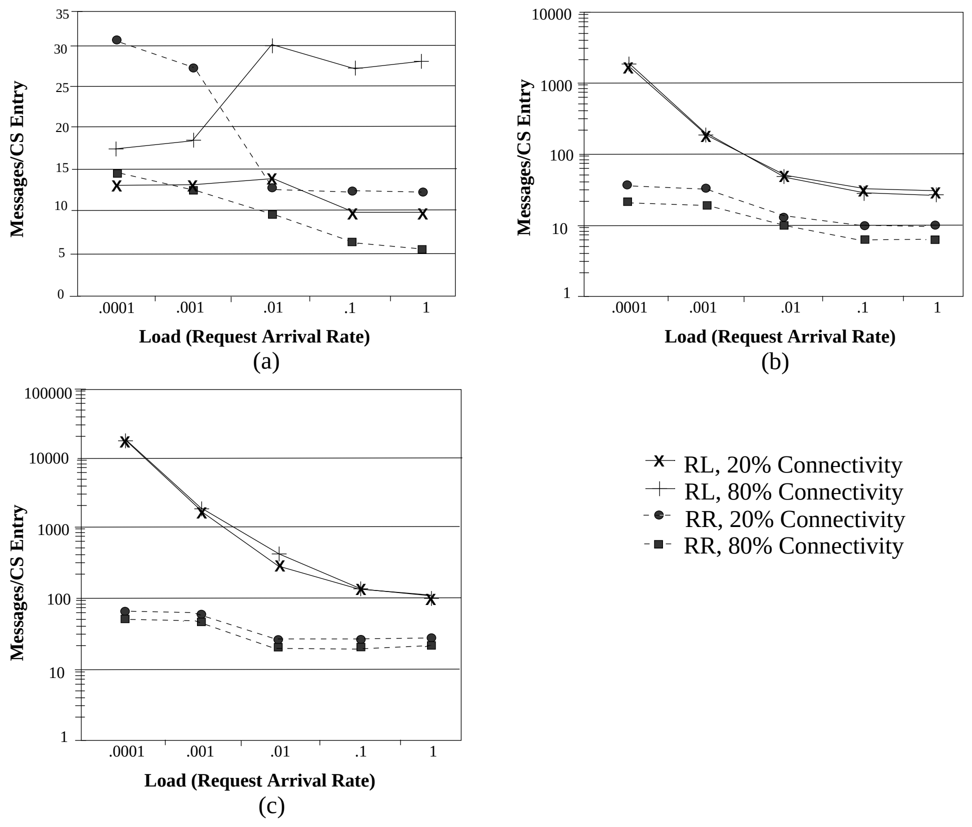

Figure 7. Load vs. Messages/CS Entry for (a) zero, (b) low, and (c) high mobility

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) connectivity for both the RL and RR simulations.

Figure 7(b) and (c) show that the RR algorithm sends fewer messages per CS entry than the RL algorithm in all simulation trials with mobility, although as load increases the message advantage of RR decreases markedly.

In all situations studied, except the RL simulation in the static case with high connectivity, the number of messages per CS entry tends to decrease as load increases. 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 becomes less pronounced as connectivity decreases. 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 nodes 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

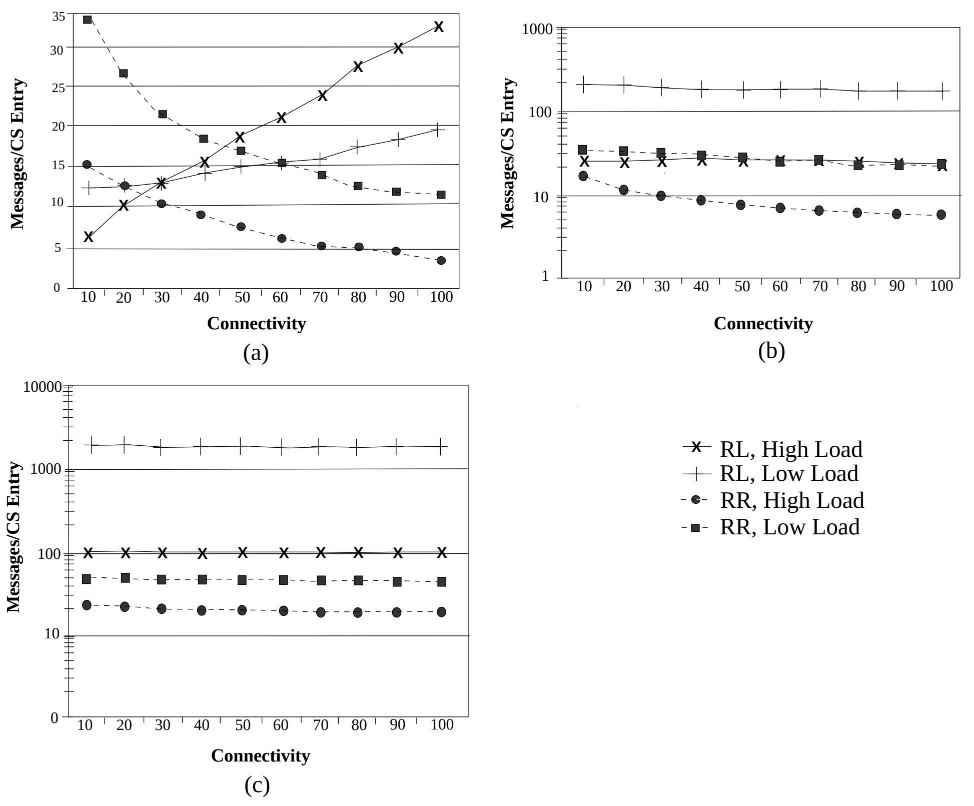

Figure 8. Connectivity vs. Messages/CS Entry for (a) zero, (b) low, and (c) high mobility

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 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 [25] at high load.

Part (a) of Figure 8 shows that in the static case the RL algorithm uses fewer messages per CS entry below 25% connectivity for high load and below 60% 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. This table includes data points from both sets of graphs depicted in this subsection. The chosen data points show average number of messages for high (80%) and low (20%) connectivity and for high and low loads in all mobility scenarios.

Table 2
Summary of messages per CS entry.

	Zero Mobility		Low Mobility		High Mobility	
	20%[a]	*80%*[a]	*20%*[a]	*80%*[a]	*20%*[a]	*80%*[a]
RR high load	13	6	11	7	30	20
RL high load	10	27	24	25	109	109
RR low load	27	13	35	20	60	50
RL low load	13	17	189	180	1900	1825

[a] Initial network connectivity.

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 [22]. In [22], 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.

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 Trans. 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 *Mobile Computing*, T. Imielinski and H. Korth, Eds., Kluwer Academic Publishers, pages 153–181, 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] M. L. Neilsen and M. Mizuno. A DAG-based algorithm for distributed mutual exclusion. In *Proc. of Intl. Conf. on Dist. Comp. Systems*, pages 354–360, 1991.

[21] E. Pagani and G. P. Rossi. Reliable broadcast in mobile multihop packet networks. In *Proc. of 3rd ACM/IEEE Intl. Conference on Mobile Computing and Networking*, pages 34–42, 1997.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.