A K-Mutual Exclusion Algorithm for Ad Hoc Mobile Networks

Jennifer E. Walter*

Abstract

A fault-tolerant distributed k-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.

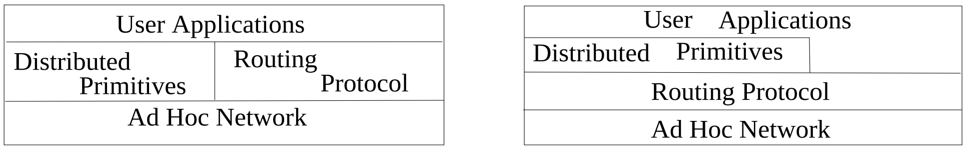
1 Introduction

In an *ad hoc* mobile network, a pair of processors communicates by transmitting messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate processors to pass the message along. Direct communication is possible only between pairs of processors 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 "formations" occur when nodes that were too far separated to communicate move such that they are within transmission range of each other. Characteristics which may distinguish wireless 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.

Routing algorithms on ad hoc networks, such as [4, 5, 7, 8, 10, 11, 12, 14, 15, 16], provide the ability to send messages from any node to any other node. 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.

In a recent paper, [20], we raised the question of whether it may be more efficient for distributed algorithms to be aware of the underlying mobility in the network (see Figure 1(a)) than to be shielded from this mobility by underlying ad hoc routing protocols (see Figure 1(b)). In that paper, we presented a mutual exclusion algorithm for dynamic ad hoc networks and tested its performance via simulation against that of a "static" mutual exclusion algorithm [18] running on top of an ad hoc routing protocol that always found shortest paths. Complexity comparison of a routing protocol is complicated by the fact that the number of messages and amount of time needed to maintain routes can be amortized over the number of applications using those routes. Therefore, to ensure the fairness of our comparison, we did not charge the routing protocol for any time or messages needed to restore shortest paths after link failures. Conversely, we did charge our algorithm for all time and messages used. Our mobility aware algorithm performed favorably in cases of zero, low and high mobility and under all loads on the system in terms of time used and compared well in most cases in terms of number of messages sent per entry to the *critical section* (CS).

*Technical Report 00-022, Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. jennyw@cs.tamu.edu. Walter is supported by GE Faculty of the Future and Dept. of Education GAANN fellowships.



(a) (b)

Figure 1: Two possible approaches for implementing distributed primitives

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. In this paper, we further test this contention, by presenting, analyzing, and testing a *mobility aware* distributed k-mutual exclusion algorithm.

The k-mutual exclusion problem involves a group of processes, each of which intermittently requires access to an identical resource or piece of code called the *critical section* (CS). At most k, $1 \leq k \leq n$, processes 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 therefore worth considering for ad hoc networks. Since wireless mobile nodes are resource poor, they may need to share resources while restricting concurrent access.

We present a k-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. Our simulation results compare the performance of this algorithm to the static distributed mutual exclusion algorithm of Bulgannawar and Vaidya [2] running on top of an ideal ad hoc routing protocol. We show that, to guarantee correctness, the algorithm of [2] requires additions to ensure that messages are delivered in the order in which they are sent between each pair of nodes and that message delivery obeys the triangle inequality[1]. We argue that these additions require a message ordering protocol in addition to the routing protocol, a requirement that makes this algorithm poorly suited to the ad hoc wireless environment, where computation requirements should consider energy limitations of the nodes.

The next section discusses related work. In Section 3, we describe our system assumptions and define the problem formally. Section 4 presents our k-mutual exclusion algorithm. A proof of correctness is presented in Section 5. Simulation results are presented in Section 6 and our conclusions are given in Section 7.

2 Related Work

Distributed k-mutual exclusion algorithms are generally grouped according to the method by which they grant access to the CS. In *permission based* algorithms, a processor requesting access to the CS must ask for and be granted explicit permission from all or some subset of the processors in the system. In *token based* algorithms, the possession of a unique token or tokens allows access to the CS. We feel that token based algorithms are a better choice for dynamic ad hoc networks because less direct inter-processor communication is required, an important consideration when link status is constantly uncertain.

Distributed algorithms for k-mutual exclusion have been presented by [2, 9, 13, 17, 19]. The algorithms of [9, 17] are permission based. Token based k-mutual exclusion algorithms are presented by [2, 13, 19].

Permission based algorithms are poorly suited to the energy poor wireless environments since they require communication between every node in the network for each access to the CS. We do not consider these algorithms any further.

[1]Messages that travel 1 hop will be received before messages sent at the same time that travel more than 1 hop.

The algorithm of [13] uses a single token with a counter to keep track of how many processes are currently in the CS. It is shown in [2] that under high load this system behaves like a single token system. Also, it requires all nodes to communicate in certain stages of the algorithm, making it too message inefficient for wireless ad hoc environments.

The algorithm of [19] uses k tokens and equates possession of any token with the ability to access the CS. This algorithm requires a node seeking access to the CS to send requests to $n-1$ other nodes in order to gain access to the CS, if it does not already hold a token. This flooding of requests makes it poorly suited to ad hoc low energy demands.

Bulgannawar and Vaidya [2] present a token based algorithm that uses k separate tokens in the system. Requests for a token follow one of k dynamic spanning trees. As requests traverse a tree from leaf to root, path compression is used to keep the tree as shallow as possible. While nodes request tokens by unique token identifiers, the algorithm allows the substitution of any token to grant access to the CS.

Each of the distributed, token based algorithms assume that the network is reliable and fully connected, allowing any processor to directly communicate with any other. These assumptions make them poorly suited to the ad hoc environment, where links form and fail as a consequence of mobility.

Our token based algorithm combines ideas from many papers. The partial reversal technique from [8], 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 k dynamic destinations. Like the algorithms of [18], [3], and [6], each processor in our algorithm maintains a request queue containing the identifiers of neighboring processors from which it has received requests for the token. Like [6], our algorithm totally orders nodes. Our algorithms maintains k tokens in the system as in [2]. When $k = 1$, the lowest node is always the current token holder, making it a "sink" toward which all requests are sent. When $k > 1$, there may be multiple sinks in the system. However, we show that all non token holding processors will always have a path to some token holding processor.

As in our 1-mutual exclusion algorithm[20], each node dynamically chooses its lowest neighbor as its preferred link to a 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 a token holder are treated symmetrically, so that requests are continually serviced while the DAG is being re-oriented and blocked requests are being rerouted. In this multiple token algorithm, it is possible for processors to receive requests while they are in the CS. If this happens, the processors may satisfy these requests immediately if they hold multiple tokens, increasing concurrent access to the CS.

3 System Assumptions

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[2] 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,

[2] 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.

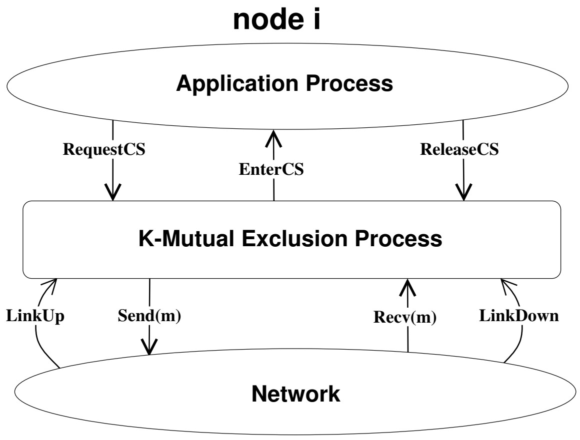


Figure 2: System Architecture

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.
- $\text{Recv}_i(j, m)$: node i receives message m from node j.
- $\text{LinkUp}_i(l)$: node i receives notification that the link l incident on i is now up.
- $\text{LinkDown}_i(l)$: 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.

- $\mathrm{Send}_i(j, m)$: 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.

$\mathrm{RequestCS}_i$, $\mathrm{EnterCS}_i$, and $\mathrm{ReleaseCS}_i$ are called *application events*, while Send_i, Recv_i, LinkUp_i, and $\mathrm{LinkDown}_i$ are called *network events*.

An *execution* e is a sequence of the form $C_0, in_1, out_1, C_1, in_2, out_2, C_2, \ldots$, where the C_j's are configurations, the in_j's are input events, and the out_j'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_j occurs at node i, then out_j and i's state in C_j are correct according to i's transition function operating on in_j and i's state in C_{j-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_j is $\mathrm{RequestCS}_i$, then the previous application event at node i (if any) is $\mathrm{ReleaseCS}_i$.
- If in_j is $\mathrm{ReleaseCS}_i$, then the previous application event at node i must be $\mathrm{EnterCS}_i$.
- If in_j 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 in_j 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.
- *k-Mutual exclusion*: If in_j is $\mathrm{EnterCS}_i$, then there are at most $k - 1$ processes in the CS. In other words, at most k processes can be in the CS at any time.

- *No starvation*: If there are only a finite number of LinkUp_i and LinkDown_i events, then if in_j is RequestCS_i, then there is a following EnterCS_i.

For the last condition, the hypothesis that link changes cease is needed because an adversarial pattern of link changes can cause starvation.

A stronger version of the no starvation requirement was suggested in [1]. This version appeals to the degree of concurrency attainable in the system and can be stated as follows:

- *No starvation with concurrency:* In any infinite extension of e with a finite number of LinkUp and LinkDown events in which $k-1$ processes are in the CS and in_j is RequestCS_i, then EnterCS_i eventually occurs even if no ReleaseCS precedes it.

  In other words, once link failures cease, if $k-1$ processors are in the CS and a processor is waiting to enter the CS, then at some later time that processor enters the CS.

4 Reverse Link (KRL) k-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 some token holder and in which every token holder has at least one incoming link.

- *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 $0 \leq i < k$, and *tokenHolder* = false otherwise.

- *totalTokens*: Number of possible tokens in the system, k.

- *numTokens*: Counter of tokens held at a node. Initially, *numTokens* = 0 if $i \geq$ *totalTokens* and *numTokens* = 1 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, $Q = \emptyset$.
- *receivedLI*[j]: Boolean array indicates whether *LinkInfo* message has been received from node j, to which a *Token* message was recently sent. Any height information received from a node j for which *receivedLI*[j] = false will not be recorded in *height*[j]. Initially, *receivedLI*[k] = 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*[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*[j] = *myHeight* for all $j \in N_i$.

4.2 Overview of the KRL 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.

We assume that $k < n$. Otherwise, there is no need for mutually exclusive access to the CS since any node can enter whenever it chooses.

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 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 calls *GiveTokenToNext*(). 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 i has more than 1 token, it calls *GiveTokenToNext*(). If Q is non-empty, i has only 1 token, 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, sets *height*[j], and increments *numTokens*. Then i lowers its height to be lower than the height of the last token holder if that height is lower than *nyHeight* at i, informs all its neighbors of its new height if necessary by sending *LinkInfo* messages, and calls *GiveTokenToNext*(). Node i always informs j of its height by sending a *LinkInfo* message to j.

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 $\leq$ what it was when i sent a *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 a 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. If i is a token holder and it just lost its last incoming link, it calls *LowerHeight*().

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 a token holder and i has no outgoing links, i calls *RaiseHeight*(). If node i is not a 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. If i is a token holder and just lost its last incoming link, it calls *LowerHeight*().

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 decrements *numTokens*, lowers *height*[*next*] to (*myHeight.h1*, *myHeight.h2* - 1, *next*), so any incoming *Request* messages will be sent to *next*, sets *tokenHolder* = false if *numTokens* = 0, and then sends a *Token* message to *next*. A *receivedLI* flag is set to false, *tokenLastSentTo* is set to *next*, and *expectedHeight* is set to *height*[*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 [8] 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 a token.

Procedure LowerHeight: Called at token holding node i when i loses its last incoming link. Uses the *partial reversal* method of [8] in lines 1-3 and informs all incoming neighbors of its height change with *LinkInfo* messages.

4.3 The KRL 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 ((|Q| > 0) and ((status = REMAINDER) or
               ((status = CRITICAL) and (numTokens > 1)))) then GiveTokenToNext()
   6.     Else // not tokenholder
   7.        If (myHeight < height[k], ∀ k ∈ N) then RaiseHeight()
   8.        Else 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.  numTokens++
   3.  height[j] := h
   4.  If (myHeight > h)
   5.     myHeight.h1 := h.h1
   6.     myHeight.h2 := h.h2 - 1   // lower my height
   7.     Send LinkInfo(myHeight) to all outgoing k ∈ N except j
   8.  SendLinkInfo(myHeight) to j
   9.  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 ∪ {j}
   2.  If ((forming[j]) and (myHeight ≠ formHeight[j])) then
   3.     Send LinkInfo(myHeight) to j
   4.  forming[j] := false
   5.  If (receivedLI[k]) 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 (tokenHolder) then
   9.     If (myHeight > height[k], for all neighbors k) then LowerHeight()
  10.  If ((myHeight < height[k], for all k ∈ N) and (!tokenHolder)) then RaiseHeight()
  11.  Else if ((|Q| > 0) and (myHeight < height[next])) then ForwardRequest()  // reroute request

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

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 ∈ N : height[l] ≤ height[j] ∀ j ∈ N
   2.  Send Request(myHeight) to next

Procedure GiveTokenToNext():  // only called when |Q| > 0
   1.  next := Dequeue(Q)
   2.  If (next ≠ i) then
   3.     numTokens--
   4.     If (numTokens = 0) then
   5.        tokenHolder := false
```

```
    height[next] := (myHeight.h1, myHeight.h2−1, next)
    receivedLI[next] := false
    Send Token(myHeight) to next
    If ((numTokens = 0) and (|Q| > 0)) then Send Request(myHeight) to next
Else  // next = i
    status := CRITICAL
    Enter CS

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

Procedure LowerHeight():
  myHeight.h1 := max_{k∈N}{height[k].h1} − 1
  S := {l ∈ N : height[l].h1 = myHeight.h1}
  If (S ≠ ∅) then myHeight.h2 := max_{l∈S}{height[l].h2} + 1
  Send LinkInfo(myHeight) to all incoming k ∈ N
```

4.4 Examples of KRL Algorithm Operation

The examples of the KRL algorithm operation on a static and dynamic network are very similar to the examples of the RL algorithm, presented in [20]. The main difference, caused by the presence of multiple tokens in the system, is that token holding nodes may lower their height when they have no incoming links.

First, the case of a static network is discussed, 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(f). 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 holders ($k = 2$) as shaded circles. The *next* pointers are shown as solid arrows. Note that when a node holds a token, its *next* pointer is directed towards itself.

In Figure 3(a), nodes 2, 3, and 4 have requested access to the CS (note that nodes 2, 3, and 4 have enqueued themselves on Q_2, Q_3, and Q_4, respectively) and nodes 2 and 3 have sent *Request* messages to node 0, which enqueued them on Q_0 in the order in which the *Request* messages were received. Node 4 sent a *Request* to node 1, since node 1 is node 4's lowest neighbor. Part (b) depicts the system at a later time, where node 1 sent a token to node 4 and has also requested access to the CS, sending a *Request* message to node 4 (note that 1 is enqueued on Q_1 and Q_4). Node 0 sent a token to node 3, following the token with a *Request* on behalf of node 2 (note that 0 is enqueued on Q_3). Observe that the logical direction of the links between node 0 and node 3 and between node 1 and node 4 change from being directed away from nodes 3 and 4 in part (a), to being directed toward nodes 3 and 4 in part (b), when nodes 3 and 4 receive *Token* messages and lower their heights. Notice also the *next* pointers of nodes 0 and 3 and nodes 1 and 4 change from both nodes 0 and 3 having *next* pointers directed toward node 0 and both nodes 1 and 4 having *next* pointers directed toward node 1 in part (a) to both nodes 0 and 3 having *next* pointers

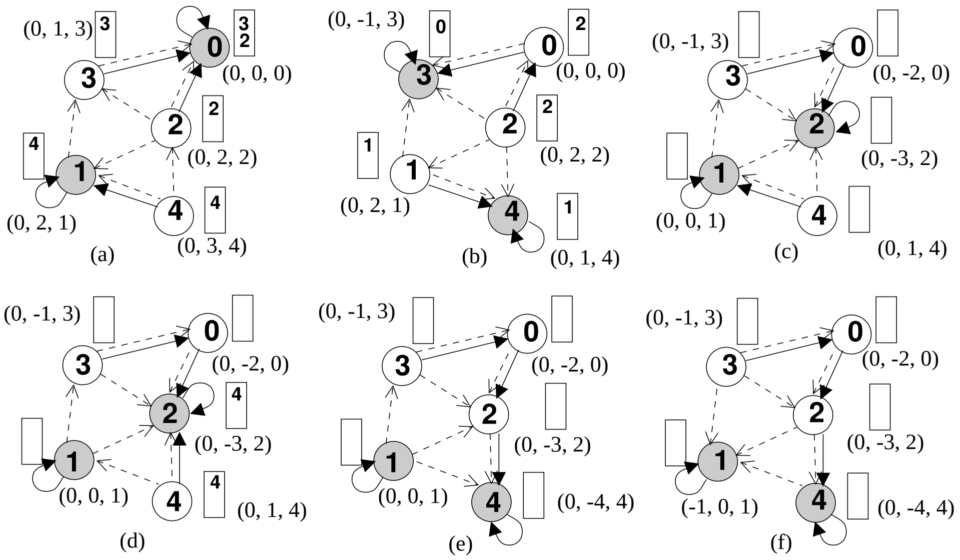


Figure 3: Operation of KRL algorithm on static network.

directed toward node 3 and both nodes 1 and 4 having *next* pointers directed toward node 4 in part (b).

Figure 3(c) shows the system configuration after node 4 has released the CS and has sent a *Token* message to node 1. Node 3 has also release the CS and has sent a *Token* message to node 0. Node 0 then sent the token to node 2. At this point in the execution there are no pending requests, as can be seen by the empty request queues.

Part (d) shows the system configuration after the host application at node 4 has made a request for CS entry and node 4 has chosen its lowest neighbor, node 2, as *next* and sent a *Request* to node 2.

In part (e), node 4 receives a *Token* message from node 2, lowers its height and enters the CS. Node 1 has received a *LinkInfo* message from node 4 and senses that it has no incoming links.

In part (f), node 1 has lowered its height to be lower than any of its neighbors, so that future requests will have a chance of being directed toward node 1.

Observe that the middle element, $h2$, of each node's *myHeight* tuple decreases by 1 for every hop the token travels, so that some token holder is always the lowest height node in the system.

Now we consider the execution of the KRL 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 node 0 by raising its height and has sent a *Request* message to node 1. Parts (d) and (e) show the system after node 0 has sent a token to node 2 and node 4 has sent a token to node 1, which then sent it to node 3.

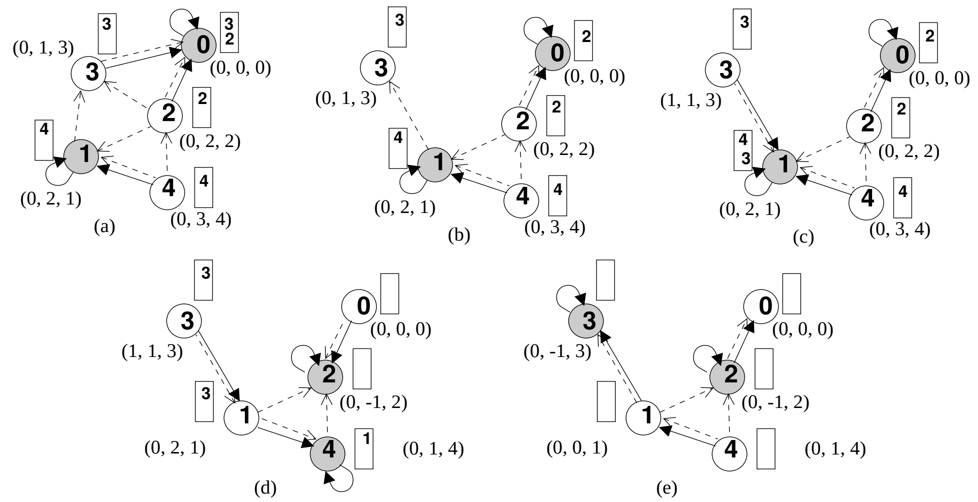


Figure 4: Operation of KRL algorithm on dynamic network.

5 Correctness of Reverse Link Algorithm

The following theorem holds because there are only k tokens in the system at any time.

Theorem 1 *The algorithm ensures k-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. Essentially, this proof is identical to the proof of correctness in [20].

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

Definition 2 *The DAG is* token oriented *in a configuration if for every node $i, i \in \{0, \ldots, n-1\}$, there exists a directed path originating at node i and terminating at a 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 [8], 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*
$(tokenHolder_i = false)$ *and* $((myHeight_i < height_i[j])$, *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 $tokenHolder_i$ = false and for which there is no directed path starting at i and ending at a 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 a 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 l raised its height infinitely often, and so on. However, Claim 1 shows that at least k nodes call *RaiseHeight*() only a finite number of times.

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

Proof: Suppose the claim is false, and some node that holds a 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 a 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, $l \neq i$, 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 [18] 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 a 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 a 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 a 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 that 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 a token and lowered its height. If the token is not held by j in C_{t-1}, then earlier j had a 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 a token holder, then there is a later configuration in which l's request chain does include a 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 a token holder. So tokens 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 a 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 some token is not eventually contained in l's request chain, there are no tokens 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: No token holder is in l's request chain. By Lemma 5, eventually some token holder will be in l's request chain.

Case 2: Some token holder is in l's request chain. Since no node stays in the CS forever, at some later time that 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.*

We have shown that the KRL algorithm ensures no starvation. The KRL algorithm does not satisfy the *no starvation with concurrency* property[1], because requests are not propagated by token holders. To see this, consider the following example: Suppose $k-1$ nodes are in the CS and another node i requests access. It can be seen from the proof of no starvation that i's request will eventually be satisfied by *some* token holder. However, if i's request reaches one of the nodes that is already in the CS, call it node j, and node j has only 1 token, then i's identifier will be enqueued at node j, not passed along to any other token holder.

A possible first step towards increasing the concurrency in the KRL algorithm is to allow node j to forward the request from node i to some other token holder if node j has any outgoing links. This has been left for future work.

The KRL algorithm does guarantee that, after link failures cease, every token holder eventually has an incoming link from some other node. This is done through the action of the *LowerHeight*() procedure.

6 Simulation Results

In this section we discuss the static and dynamic performance of the k-Reverse Link (KRL) algorithm compared to a k-mutual exclusion algorithm designed to operate on a static network. We simulated Bulgannawar and Vaidya's token based k-mutual exclusion algorithm [2] 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 "Bulgannawar/Vaidya with routing" (BVR). The Bulgannawar and Vaidya algorithm was used because it requires less communication than other token based k-mutual exclusion algorithms. Also, this algorithm 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 Bulgannawar and Vaidya's algorithm: that it always provides shortest paths and its time and message complexity are zero. If our simulation shows that the KRL algorithm is better than the BVR combination in some scenario, then the KRL algorithm will also be better than Bulgannawar and Vaidya's algorithm in that scenario when *any* real ad hoc routing algorithm is used. If our simulation shows that the KRL algorithm is worse than the BVR 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.

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. Note that a clique on 30 nodes has 435 (undirected) links. In the graphs of the results in this section, each plotted point represents the average of five repetitions of the simulation. Thus, in plots of average time per CS entry, each point is the average of the averages from five executions, and similarly for plots of average number of messages per CS entry.

The same set of initial graphs were used on both the KRL and BVR simulations for each experiment. During periods of mobility, link changes were not allowed to change the percent connectivity of the initial graph more than 10% in either the positive or negative direction.

Briefly, and without getting into the details and intricacies of the algorithm, the BVR simulation works as follows: When host applications request access to the CS, nodes send requests for a particular token to some other node which held that token in the recent past. In this algorithm both tokens and nodes have unique identifiers. Each node maintains a request queue as in the KRL algorithm. Additionally, in BVR, a *pointer*[t] array for each token t contains the identifier of the node to which a request for t will be sent. When a node issues a request for a particular token, it saves the identifier of that token in a variable called *waiting_for_token*. Each node uses some heuristic to choose the identifier of the token for which it will send a request. Request messages carry the identifier of the requesting node and the token which is being requested. All incoming requests at a node I for token $t = \mathit{waiting\_for\_token}$ are enqueued in a request or "node queue" at I. All incoming requests for token $t \neq \mathit{waiting\_for\_token}$ are forwarded to *pointer*[t] and nothing is enqueued. Each token t carries a "token queue", to which each node appends its node queue when it receives t. After appending its node queue to a token queue, a node deletes all entries on its node queue. When node I exits the CS holding token t, if t's token queue is empty, I sends an $\mathit{Inform}(I, t)$ message to ν other nodes, where ν is a performance parameter.

The BVR simulation starts with nodes with identifiers ranging from 0 to $k-1$ in possession of the tokens and has each node set k pointers to the holders of the tokens (including a pointer to itself if it holds a token). The algorithm maintains a set of k dynamic spanning trees rooted at the token holder. As the tokens move, all nodes continually modify their pointers to point to the most recent node that held a particular token from the information they have at that point in the execution. Since this algorithm gives each token a unique identifier in the range 0 through $k-1$, each token is initially positioned at the node whose identifier matches the token identifier. For the BVR simulations, a connected graph whose initial edges were chosen at randon with the desired number of links was read from a file and then the pointers at each node were set to form k spanning trees of depth 1 rooted at each of the k token holders. Shortest paths from all nodes to the roots of the virtual spanning trees were calculated next. After this, the mutual exclusion algorithm was started and counting messages and waiting time per CS entry was initiated. When link changes occurred, the time or messages needed to recalculate shortest path routes in the modified graph were not counted. Any added time and distance that the application messages traveled due to route changes were counted, charging one message per link traversed.

The parameter ν represents the number of nodes to which a token holder sends messages if it has no requests pending in BVR. During simulations, ν was set to 2, and the number of tokens, k, used for each set of simulations was 3, as in [2], so that comparisons could be made with their results. The heuristic used in the BVR simulation for choosing a token was the "last seen token" heuristic [2], in which the token requested is either the last token received at a node or the last token for which a node received an *Inform* message.

The KRL simulation also starts with nodes with identifiers ranging from 0 to $k-1$ holding tokens. If any token holder initially had no incoming links, we initially adjusted its height so it had at least one incoming link. For simulations of KRL, a connected graph whose initial edges were chosen at random with the desired number of links was read from a file, node heights and link directions were initialized, and then the algorithm and performance measurements were started. 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 Comment on correctness of BVR simulation

While simulating the performance of BVR when nodes are mobile, it was discovered that nodes may starve when message delays are highly asynchronous. The delay of the *Inform* messages in the network is the cause of the observed starvation problem. Starvation of nodes is probably due to the reliance of this algorithm on message delivery obeying the triangle inequality. The triangle inequality requires that messages traveling one hop must be received before messages sent at the same time (or later) that travel more than one hop.

To see why nodes may starve in BVR, consider the following counter-example, a scenario occurring sometime after the BVR execution has started. Figure 5(a)-(f) depicts four nodes in a larger system, with time increasing from part (a) to part (f). The current holder of token x is shaded in each part of this figure and the token queue is shown as a shaded rectangle, the arrows represent the value of *pointer*$[x]$, and the unshaded rectangles show the node queues at each of the four nodes in each part of the figure. No other tokens but x are held by any node depicted in Fig. 5.

In Figure 5(a), node A exits the CS holding token x, but the token queue of x is empty so node A sends an *Inform*(A, x) message to node B. Some time after A sends the *Inform* message, nodes A, B, and C receive *Request*s for token x, node A sends token x to node C, and node A sets *pointer*$[x] = C$, as shown in Figure 5(b). Figure 5(b) shows the configuration immediately after node C receives token x. Note that node C has enqueued B's identifier on the token queue of token x and node B has enqueued node D's identifier on its node queue. Node C then enters and exits its CS, sends token x to node B, and sets *pointer*$[x]$ to B. Node B receives token x in Fig. 5(c) and appends its node queue to the token queue of x. However, *node B has not yet received the Inform(A, x) message from node A.*

After entering and exiting its CS, node B sends token x to node D and sets *pointer*$[x]$ to D in Figure 5(d). After this, node B receives the delayed *Inform*(A, x) from node A (Figure 5(e)), at which point B sets *pointer*$[x] = A$. Now a cycle exists in the structure created by the *pointer*$[x]$s at nodes A, B, and C, as shown in Fig. 5(e).

In figure 5(f), the host application at node A has requested access to the CS, choosing token x, and has sent a *Request*(A, x) message to node C. Node C has received the request from A and has forwarded a

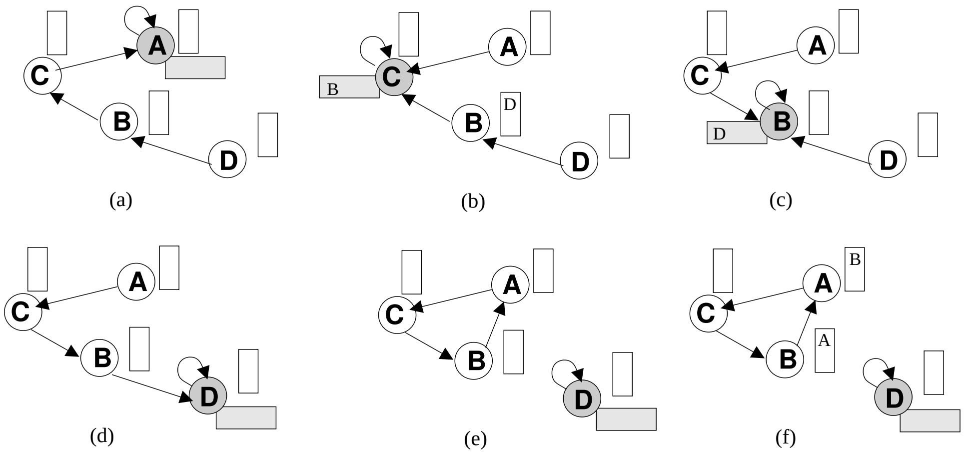


Figure 5: Example of bad BVR simulation execution.

Request(A, x) message to node B. At roughly the same time that node C was forwarding the request from node A, the host application at node B requested access to the CS, choosing token x, and node B sent a *Request*(B, x) message to node A. The result of these events is that node A enqueues B's identifier on its node queue and node B enqueues A's identifier on its node queue. Neither node A nor B will receive token x because they will not send another request for any token while they are waiting to receive token x (although they may forward requests containing the identifiers of other nodes for tokens other than x.) No future requests containing the identifier of A or B will be sent, so the identifiers of nodes A and B will not be added to any token queues. Therefore, since each token is sent only to nodes whose identifiers are enqueued on its token queue, neither A nor B will receive any token, resulting in starvation.

One way to correct this starvation problem would be to include time stamps on each message sent so that nodes could detect and ignore old *Inform* messages. However, this would require additional hardware on the mobile nodes and would also increase the message size, requiring more energy to be used in each transmission.

6.2 Average waiting time per CS entry

Figure 6 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^{-3} (the mean time units between requests is 10^3) 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% connectivity (348 links).

As shown in Figure 6(a), the performance of the BVR and KRL simulations is roughly equal at the lowest loads for both connectivity levels and at the highest load for 20% connectivity. The decrease in average waiting time at the highest load for the KRL simulation at both connectivity levels and the BVR simulation at 20% connectivity is caused by an extremely unfair pattern of token use. For example, in a particular execution, two nodes entered the CS about 50000 times during the same time period in which the rest of the nodes entered the CS only about 1500 times. This skewed pattern of token use results in zero wait time for the majority of CS entries, bringing down the average wait time at the highest load. The tokens are used more fairly at loads below 1 in KRL, probably because the initial request paths are modified as tokens move at lower load, whereas at the highest load, nodes request so frequently that some tokens continually traverse long circuitous request paths, while others traverse shorter paths or never move

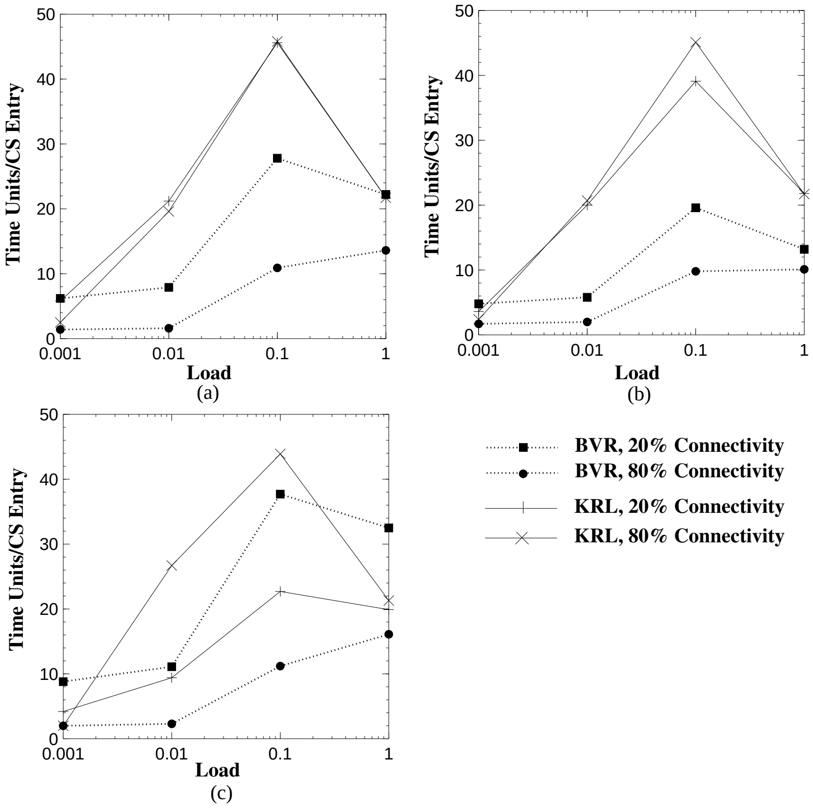


Figure 6: Load vs. time/CS entry for (a) zero, (b) low, and (c) high mobility ($k = 3$).

at all, depending on the initial DAG and the links requests are initially sent over.

Tokens tended to be used more fairly in the BVR simulation at the highest load and 80% connectivity, with average wait time increasing as load increased. The BVR simulation using three tokens for 30 nodes had an average wait time at the highest load which was roughly one third of that observed in the RR and RL simulations in [20]. This probably represents a fairly round-robin pattern in which each token was used by about one third of the nodes in the system. The fact that, in the BVR simulation, the nodes initially choose a token identifier at random may account for the resulting fair usage of the tokens throughout the simulation.

In part (a) of Figure 6, the static case, the decrease in average time at 80% connectivity in BVR as load decreases corresponds well to the results shown for average time per CS entry in [2]. The slight differences are caused by the fact that Bulgannawar and Vaidya plotted runs with 100% connectivity and the runs shown here are plotted with 80% connectivity.

Parts (b) and (c) of Figure 6 show that the performance of the KRL algorithm, in terms of average wait time per CS entry, improves at 20% connectivity when link failures cause token circulation to change. For the BVR simulation, higher connectivity always results in lower average wait time than lower connectivity. At the highest mobility, the KRL simulation running on a network with 20% connectivity outperformed the BVR simulation at 20% connectivity for all loads tested.

Figure 7 plots the average number of time units elapsed between host request and subsequent entry to the CS 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 number of time units between requests is 10^3) and high load, where $\lambda_{req} = 1$ (the mean number of time units between requests is 1) for both the BVR and KRL simulations.

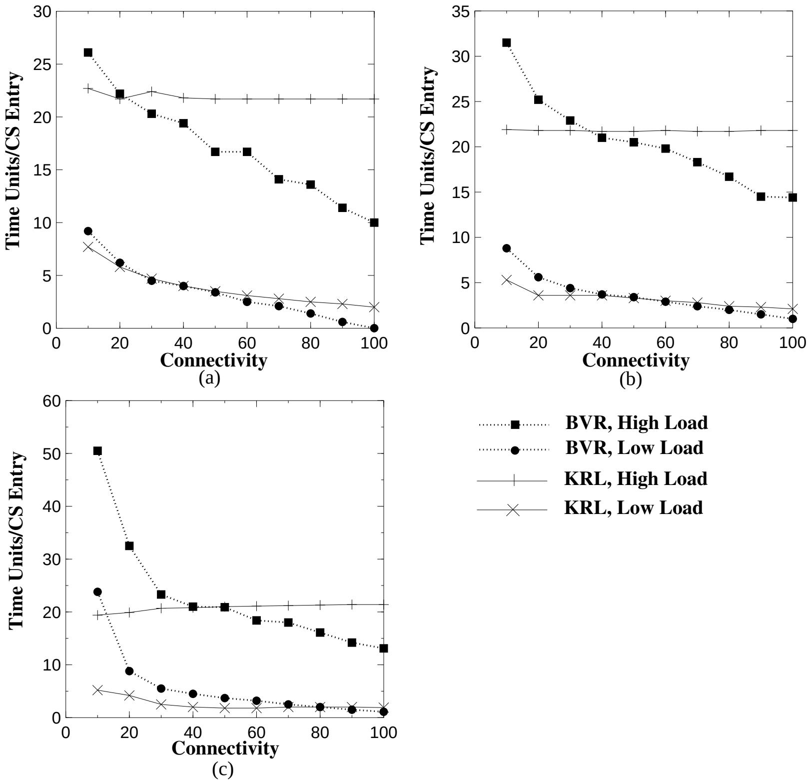


Figure 7: Connectivity vs. time/CS entry for (a) zero, (b) low, and (c) high mobility ($k = 3$).

In the BVR simulation, performance in terms of average waiting time per CS entry, improves as connectivity increases in all mobility scenarios. In the static case, Figure 7(a), the average waiting time of 10 time time units at 100% connectivity and high load compares well with the average 9 time units per CS entry cited in the results of Bulgannawar and Vaidya[2].

The performance of the KRL algorithm at high load is, for the most part, unaffected by changing connectivity, except for a slight increase in average wait time under high mobility as connectivity increases.

Overall, increasing mobility is slightly more detrimental to the average wait time per CS entry in the BVR simulation, as shown in Figure 7(b) and (c). At the highest mobility tested, the KRL simulation performed better at low load below a connectivity of 70%, and better below a connectivity of 40% at high load.

The results of the simulations measuring time per CS entry are summarized in Table 1.

Table 1: Summary of time per CS entry in k-mutual exclusion simulations.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
BVR high load	22	14	15	17	33	16
KRL high load	22	22	22	22	20	21
BVR low load	6	1	30	2	9	2
KRL low load	6	3	4	2	4	2

6.3 Average messages per CS entry

In the BVR simulation, *Request* and *Token* messages are sent along a dynamic "virtual forest" of token pointers. *Inform* messages are sent to keep tokens in circulation and to make nodes aware of the current token positions by changing the pointers in the virtual forest. Each message from a node to its virtual neighbor is converted by the underlying routing protocol into a sequence of actual messages that traverse the current shortest path from the sender to the recipient. Since all nodes are considered to be the virtual neighbors of all other nodes at all times, messages always travel the current shortest path provided by the routing protocol.

The KRL simulation sends *Request* and *Token* messages along the actual token oriented DAG. As tokens move, each node receiving a token sends *LinkInfo* messages to all its outgoing neighbors and to the token sender. 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 hops taken by algorithm messages and *Inform* messages to update the virtual forest of pointers for BVR, versus the relative number of hops taken by algorithm messages and *LinkInfo* messages to maintain the DAG for KRL. When interpreting these results, it is important to remember that *the simulations of the BVR algorithm is not charged for messages used to find new physical routes between pairs of nodes as topology changes.* Therefore, if the KRL simulation sends fewer messages in some situation, it will certainly send fewer messages when the BVR simulation is charged for finding routes, even if the number of messages sent can be prorated for BVR. Even in situations where the BVR simulation sends fewer messages than does KRL, it may be the case that it would send more messages than KRL if it were charged for finding routes.

Figure 8 plots the average number of messages received per CS entry against values of λ_{req} ranging from 10^{-3} (the mean number of time units between requests is 10^3) to 1 (1 mean time unit between requests) 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 BVR and KRL simulations.

From Figure 8, we can see that the BVR simulation tends to send fewer messages as load increases at both connectivity levels tested. In the static case, part (a), the KRL simulation shows similar behavior to the RL algorithm from [20].

In part (a) of Figure 8, the static case, the average number of messages decreases as load increases at 80% connectivity, the same pattern reported in [2]. The explanation for the differences in values of points plotted here and those plotted by Bulgannawar and Vaidya is that the runs shown here are plotted with 80% connectivity, while theirs were plotted with 100% connectivity.

As mobility increases, as shown in Figure 8 (b) and (c), the number of messages sent by the BVR simulation increases only slightly. The number of messages sent is greater at 20% connectivity than it is at 80% connectivity in the BVR simulation in all scenarios due to the potentially larger number of hops between any particular pair of nodes with sparser connectivity. For the KRL simulation, the number of messages sent to maintain the DAG increases as mobility increases, becoming very large at low load, when the majority of messages are sent to maintain the DAG.

For the KRL simulation, the number of messages per CS entry compares favorably to the BVR simulation in the static case, shown in Figure 8(a), at both 20% and 80% connectivity. With low mobility, part (b), KRL sends only about as many messages as the BVR simulation when load is higher than 0.01. Under high mobility, part (c), the KRL simulation sends approximately the same number of messages as BVR for loads higher than 0.1.

Figure 9 shows the results of experiments designed to understand the effect of connectivity on the number of messages per CS entry. In this 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

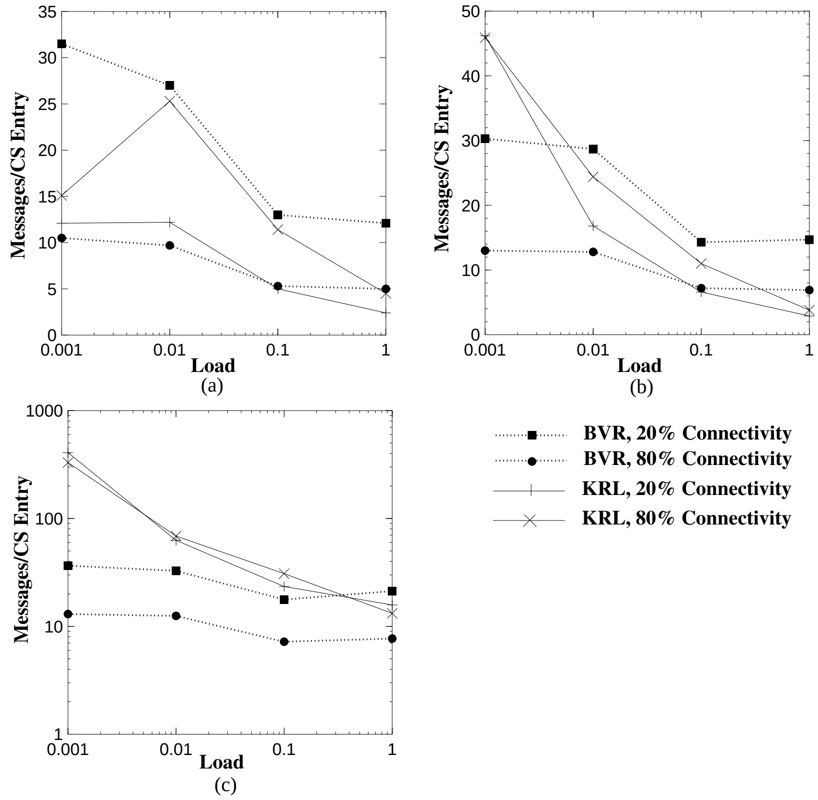


Figure 8: Load vs. messages/CS entry for (a) zero, (b) low, and (c) high mobility ($k = 3$).

axis. Curves are plotted for low load ($\lambda_{req} = 10^{-3}$), where the mean number of time units between requests is 10^3, and high load ($\lambda_{req} = 1$), where the mean number of time units between requests is 1, for both the BVR and KRL simulations.

In Figure 9(a), the static case, the KRL simulation has the lowest number of messages per CS entry below connectivity of 80%. This is due to the fact that all the tokens do not circulate at high load. One or two token holders get tokens disproportionately often, while the other token is used by the majority of the nodes. When certain tokens do not circulate, the node possessing the token can enter the CS every time its host makes a request. When this happens continually, few or no messages are sent either for requests or for rearranging the DAG as that token circulates in a very small portion of the network or not at all, bringing down the overall average. As mobility increases (Figure 9(b) and (c)), the tokens are circulated more fairly in the KRL simulation at high load, since the DAG adjusts to link failures.

In the static case, Figure 9(a), the average number of messages at 100% connectivity and high load (2.5) compares well with the average messages per CS entry cited in the results of Bulgannawar and Vaidya[2] (about 3).

The results of the simulations measuring messages per CS entry are summarized in Table 2.

6.4 Performance of KRL when number of tokens varies

To determine how the performance of the KRL simulation varied as the number of tokens in the system varies, we ran experiments with 1, 3, and 5 tokens on a 30 node system. All other parameters are identical to those described in the previous simulations discussed in this chapter.

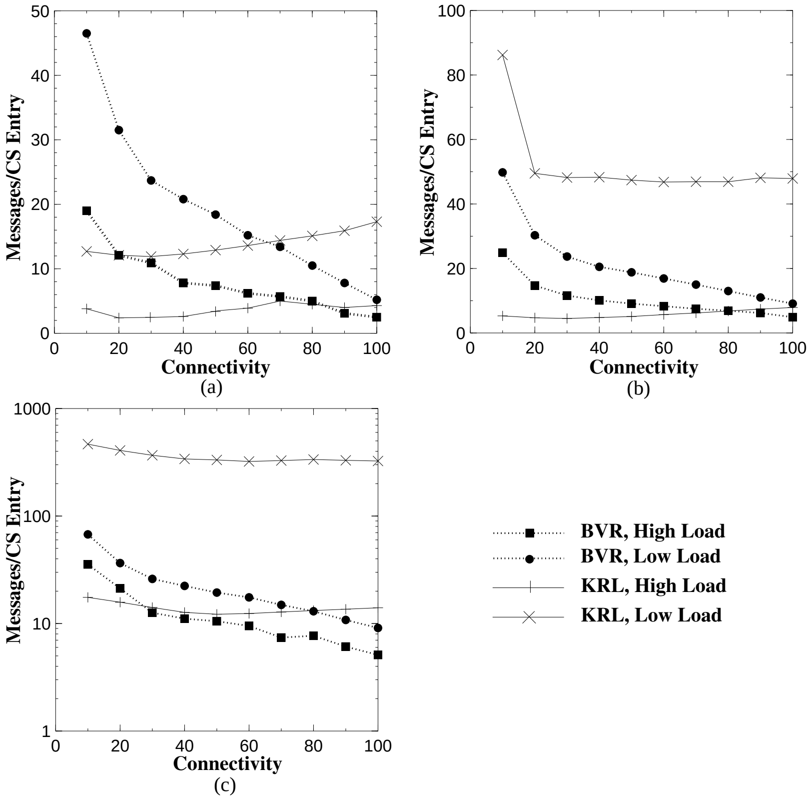


Figure 9: Connectivity vs. messages/CS entry for (a) zero, (b) low, and (c) high mobility ($k = 3$).

Table 2: Summary of messages per CS entry in k-mutual exclusion simulations.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
BVR high load	12	5	15	7	21	8
KRL high load	2	5	5	7	16	13
BVR low load	32	11	30	13	37	13
KRL low load	12	15	50	47	408	336

As would be expected, the performance of the KRL simulation with 1 token is the same as the performance of the RL algorithm from [20]. In general, the following graphs and tables show that performance improves with increasing number of tokens.

6.4.1 Average time units per CS entry

Figure 10 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^{-3} (the mean time units between requests is 10^3) 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% connectivity (348 links) for runs of the KRL simulation with 1, 3, and 5 tokens.

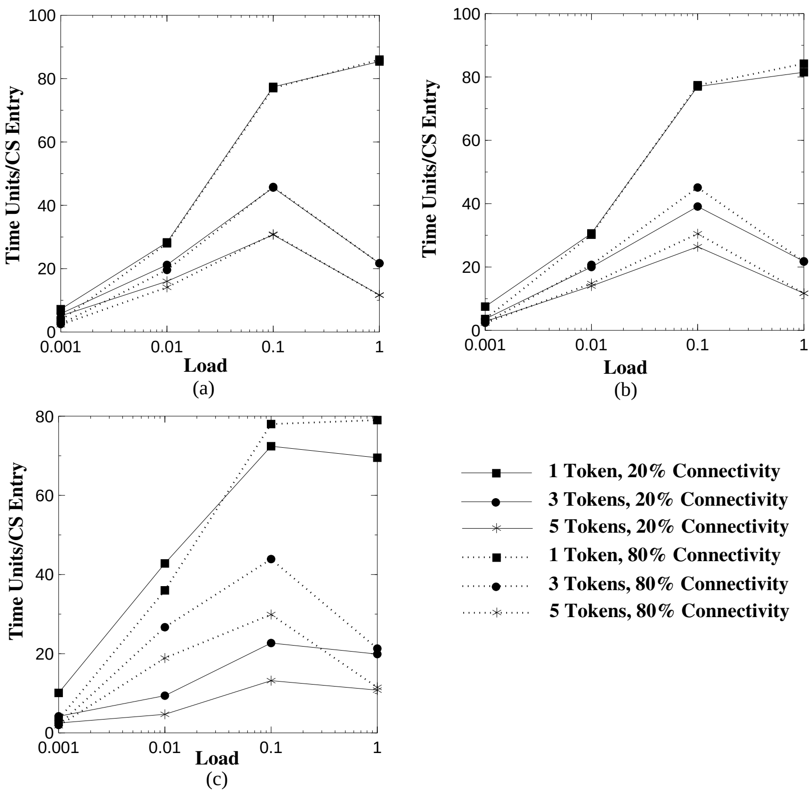


Figure 10: Load vs. time/CS entry (a) zero, (b) low, and (c) high mobility.

Figure 10 shows that, as mobility increases, the advantages of multiple tokens in the system, in terms of average waiting time per CS entry, are realized only at medium loads (i.e., loads from about 0.01 to 0.1). At the lowest and highest loads, there was little or no difference in the average waiting time per CS entry between the 3 and 5 token experiments at the same connectivity level. It is probably the case that a greater difference would be shown, particularly at high load, if the number of tokens was varied more widely.

In Figure 10, parts (b) and (c), note that the average time per CS entry increases as the load increases from 0.001 to 0.1 at both connectivity levels, in all simulations. But at the highest rate of mobility, in the multiple token simulations, there is a peak in the average time per CS entry at a load of 0.1 and 80% connectivity that is not apparent at the 20% connectivity level. When the load increases to 1, both connectivity levels have roughly the same average time per CS entry. The increase in mobility may actually be ensuring more fair patterns of token use at 20% connectivity than at 80% connectivity. Since request paths are broken more often with lower connectivity, the tokens may be allowed to circulate among all nodes more fairly at 20% connectivity. At 80% connectivity and load below 0.1, nodes may become aware of other tokens in their vicinity (besides the one they last requested) through the propagation of *LinkInfo* messages, allowing the circulation of the tokens to larger portions of the network. At the highest load, nodes request so often that token movement may be much more restricted, particularly at high connectivity. This may be due to the fact that at high connectivity and high load, nodes send requests out on the last link over which they received the token before *LinkInfo* messages can inform them of closer tokens. This effect is more pronounced at high connectivity because the last link a node sent a request on is much less likely to be broken between consecutive requests. This may cause some tokens to circulate in very small corners of the network, lowering the overall time per CS entry at higher connectivity.

Figure 11 plots the average number of time units elapsed between host request and subsequent entry to the CS 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 number of time units between requests is 10^3) and high load, where $\lambda_{req} = 1$ (the mean number of time units between requests is 1) for the KRL simulation with 1, 3, and 5 tokens.

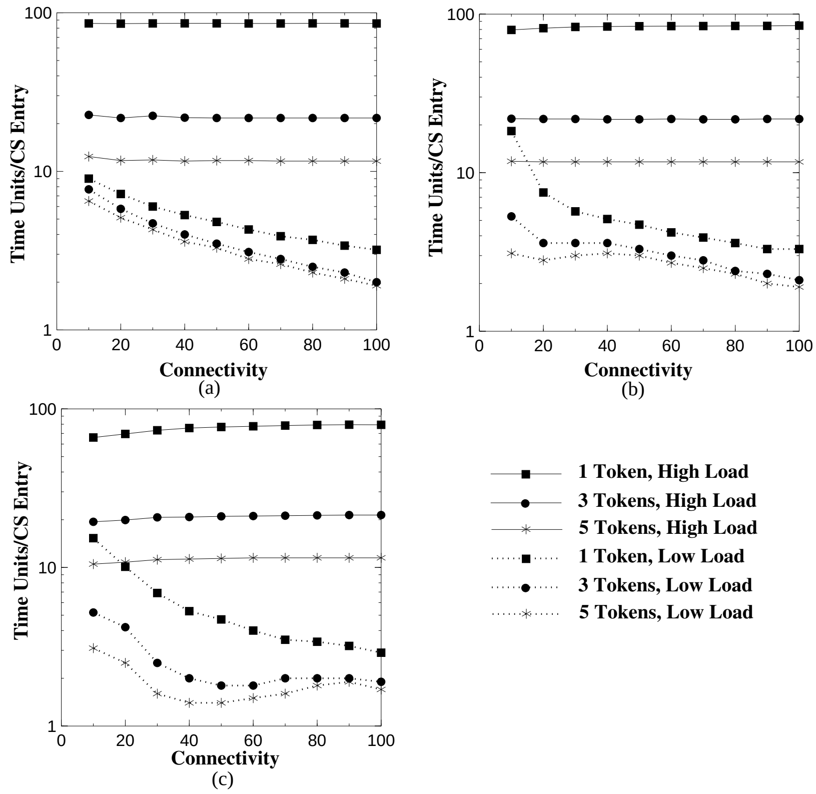


Figure 11: Connectivity vs. time/CS entry (a) zero, (b) low, and (c) high mobility.

As can be seen in Figure 11, the performance improvement gained when more tokens are in the system is greater, in most cases, when load on the system is highest.

Figure 11 indicates that connectivity does not affect the waiting time per CS entry in the KRL simulation at high load. At high load, the KRL algorithm does not exploit connectivity even when nodes are moving, regardless of the number of tokens in the system. When load is high, the KRL 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 KRL, connectivity does affect the waiting time per CS entry because request messages are not always sent over the path last traveled by the token. 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, particularly when connectivity is highest.

In part (c) of Figure 11, the low load curves for 3 and 5 tokens dip at around 40% connectivity, rising again at around 80% connectivity. We speculate that some qualitative behavior of the KRL algorithm with at least 3 tokens on a 30 node graph changes when connectivity increases through this range, but we are currently uncertain of the cause of this behavior.

The results of the simulations measuring time per CS entry are summarized in Table 3.

Table 3: Summary of time per CS entry for 1, 3, and 5 tokens in the KRL simulation.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
1 token, high load	85	86	82	84	70	79
3 tokens, high load	22	22	22	22	20	21
5 tokens, high load	12	12	12	12	11	12
1 token, low load	7	4	7	4	10	4
3 tokens, low load	6	3	4	2	4	2
5 tokens, low load	5	2	3	2	3	2

6.4.2 Average messages per CS entry

Figure 12 plots the average number of messages received per CS entry against values of λ_{req} ranging from 10^{-3} (the mean number of time units between requests is 10^3) to 1 (1 mean time unit between requests) from left to right along the x axis. Plots are shown for runs with 20% (87 links) and 80% (348 links) connectivity for the KRL simulation with 1, 3, and 5 tokens.

In Figure 12(a), there is a peak in average number of messages sent at a load of 0.01 with 80% connectivity, a pattern that is not apparent at 20% connectivity. An explanation for this peak may be that token movement more effectively shortens request path length at high connectivity with load below 0.01. When load is 0.001, there appears to be sufficient time between requests for nodes to receive *LinkInfo* messages sent as the token moves, providing the opportunity for nodes to send requests over direct links to the token holder rather than over the last link on which they sent the token. Also, at loads as low as 0.001, nodes retain the token between requests more often than they do at a load of 0.01. When nodes retain the token between requests, no messages are sent either for consecutive requests or for token movement between those requests. At loads higher than 0.01, the token is used more often in a given number of hops because more nodes request the CS simultaneously, resulting in a lower overall number of messages per CS entry.

Figure 13 shows the results of experiments designed to understand the effect of connectivity on the number of messages per CS entry. In this 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 ($\lambda_{req} = 10^{-3}$), where the mean number of time units between requests is 10^3, and high load ($\lambda_{req} = 1$), where the mean number of time units between requests is 1, for the KRL simulation with 1, 3, and 5 tokens.

In the static case, Figure 13(a), it is interesting to note that the linear increase in the average number of messages sent with high load as connectivity increases in a 1 token system does not occur when there are multiple tokens. In simulations with 1 token, this linear increase is due to the greater number of *LinkInfo*

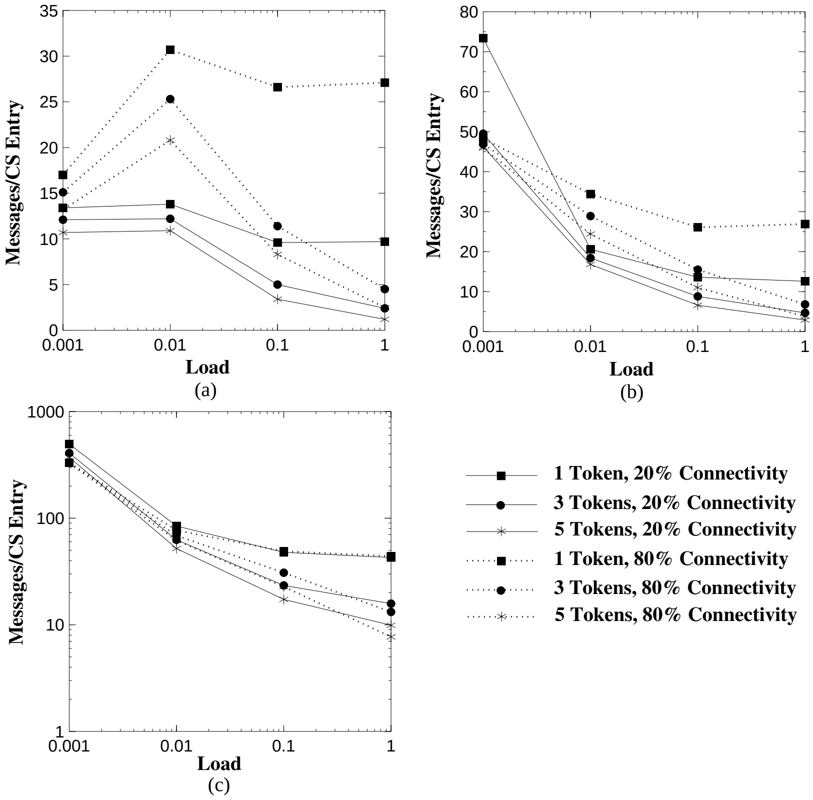


Figure 12: Load vs. messages/CS entry (a) zero, (b) low, and (c) high mobility.

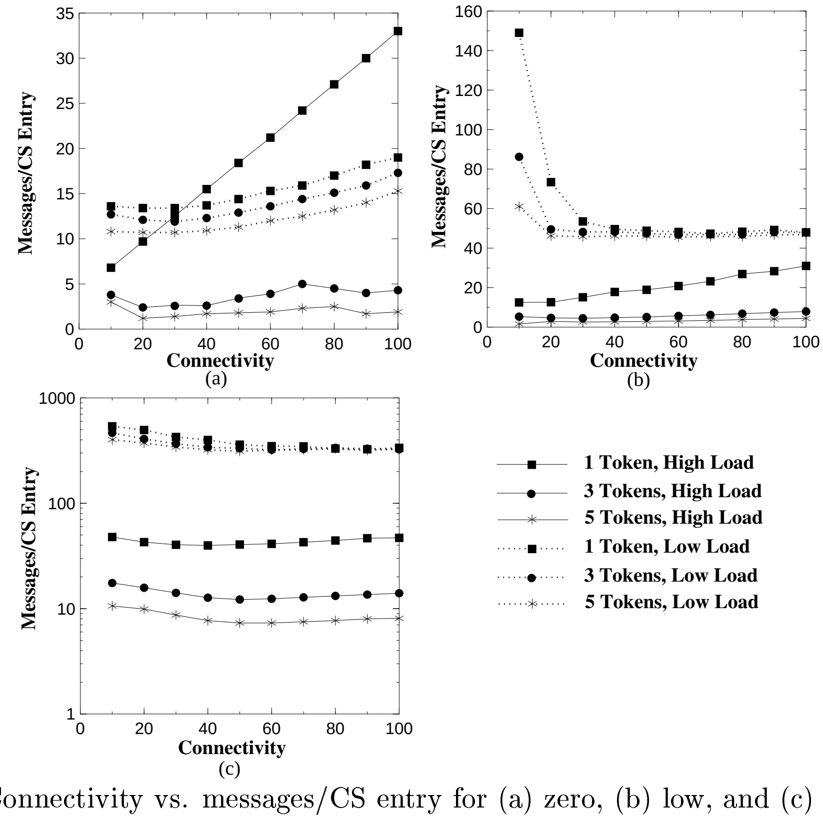


Figure 13: Connectivity vs. messages/CS entry for (a) zero, (b) low, and (c) high mobility.

messages sent as the token moves and connectivity increases. In the multiple token case, the tokens are not used fairly when nodes are not mobile. This is because request paths are formed and may not be changed as execution continues, particularly at low connectivity. The result is that some tokens are used by a few nodes very frequently, causing few or no messages to be sent for these tokens and allowing a few nodes to make tens of thousands of CS entries while the rest of the nodes make only thousands of CS entries. When few or no messages are sent for a large percentage of CS entries, the overall average is reduced.

As can be seen in Figure 13, the performance improvement gained when more tokens are in the system is greater, in most cases, when load on the system is highest.

The results of the simulations measuring time per CS entry are summarized in Table 4.

Table 4: Summary of messages per CS entry for 1, 3, and 5 tokens in the KRL simulation.

	Zero Mobility		Low Mobility		High Mobility	
Connectivity	*20%*	*80%*	*20%*	*80%*	*20%*	*80%*
1 token, high load	10	27	13	27	43	44
3 tokens, high load	2	5	5	7	16	13
5 tokens, high load	1	3	3	4	10	8
1 token, low load	13	17	73	48	496	331
3 tokens, low load	12	15	50	47	408	336
5 tokens, low load	11	13	46	47	373	328

7 Conclusion and Discussion

We have presented a mobility aware k-mutual exclusion algorithm for mobile ad hoc networks. We proved that this algorithm provides mutually exclusive access to a critical section for up to k nodes at a time and that every request will eventually be satisfied if link failures cease.

We compared the performance of this algorithm to a distributed k-mutual exclusion algorithm[2] running on top of an ad hoc routing protocol that always provides shortest paths at no cost. Our algorithm performed favorably in terms of average time per CS entry, particularly when nodes were mobile. The performance of our algorithm at high load with mobility, in terms of number of messages sent per CS entry, suggests that it may be more efficient than running a static algorithm with a routing protocol when the routing protocol is charged for time and messages used.

Future work will include the inclusion of a mechanism which allows nodes to forward requests received while they are in the CS, thereby increasing concurrent token usage. Also, we hope to devise better methods of measuring the degree of concurrency attained in simulations.

References

[1] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 18(3):939–953, 1994.

[2] S. Bulgannawar and N. H. Vaidya. A distributed k-mutual exclusion algorithm. In *Proc. of 15th IEEE Intl. Conf. on Distributed Computing Systems*, pages 153–160, 1995.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] S. T. Huang, J. R. Jiang, and Y. C. Kuo. K-coteries for fault-tolerant k entries to a critical section. In *Proc. of IEEE Intl. Conference on Distributed Computing Systems*, pages 74–81, 1993.

[10] 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.

[11] 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.

[12] 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.

[13] K. Makki, P. Banta, K. Been, N. Pissinou, and E. K. Park. A token based algorithm for distributed k mutual exclusion. In *Proc. of 4th IEEE Symp. on Parallel and Distributed Processing*, pages 408–411, 1992.

[14] 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.

[15] 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.

[16] 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.

[17] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(1989):189–193, 1989.

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

[19] P. K. Srimani and R. L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1992):51–57, 1991.

[20] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. Accepted to *Telecommunication Systems Journal, special issue on 1998 DialM papers*, 2000.