A Token Forwarding K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks

Jennifer E. Walter Guangtong Cao Mitrabhanu Mohanty
Computer Science Department, Texas A&M University, College Station, TX 77840-3112
e-mail: {jennyw,g0e7670,mmohanty}@cs.tamu.edu

Abstract

A fault-tolerant token based distributed $k$-mutual exclusion algorithm which adjusts to node mobility is presented. The algorithm requires nodes to communicate with only their current neighbors, making it well-suited to the ad hoc environment. A “token forwarding” modification to the basic algorithm is shown to lower the time each node waits to enter the CS by circulating unused tokens among participating processors. Further heuristic modifications are shown to improve the message complexity of the algorithm.

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, limited energy supplies, and highly variable message delays. These characteristics make ad hoc networks challenging environments in which to implement distributed algorithms.
Related work on distributed algorithmic development for ad hoc networks includes numerous routing protocols (e.g., [8, 9, 11, 13, 18, 20, 21, 25, 26, 27]), wireless channel allocation algorithms (e.g., [14]), leader election algorithms [15, 23], and protocols for broadcasting and multicasting (e.g., [8, 12, 24, 30]). Dynamic networks are fixed wired networks that share some characteristics of ad hoc networks, since failure and repair of nodes and links is unpredictable in both cases. Research on dynamic networks has focused on total ordering [19], end-to-end communication, and routing (e.g., [1, 2]).

The $k$-mutual exclusion problem involves a group of $n$ 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.

The contribution of this paper is a generalization of the 1-mutual exclusion algorithm presented in [32] to a topology sensitive distributed $k$-mutual exclusion algorithm (called the KRL algorithm), which induces a logical directed acyclic graph (DAG) on the network, dynamically modifying the logical structure to correspond to the actual physical topology in the ad hoc environment. The algorithm ensures that all requesting processors eventually gain access to the CS once the network stabilizes and communication links are reestablished. The KRL algorithm includes a new token forwarding technique, designed to promote token circulation throughout the network. This modification to the basic KRL algorithm is shown to decrease the time per CS entry at each node to nearly half of the time taken in the basic algorithm under particular loads on the system by continuously forwarding tokens throughout the network. A further heuristic modification to the basic KRL algorithm, in which nodes hold unused tokens prior to forwarding them, is shown to decrease the number of messages used in the token forwarding algorithm by more than half when load on the system is low.

The next section discusses related work. In Section 3, we briefly describe our system assumptions and problem statement. Section 4 describes the $k$-mutual exclusion algorithm. A sketch 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 classified according to the method by which they grant access to the CS. In permission based algorithms (e.g., [16, 28]), 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 (e.g., [5, 22, 31]), 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.

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

The token based 1-mutual exclusion algorithm of [32], from which the algorithm we present was adapted, provides a synthesis of ideas from several papers. The partial reversal technique from [13], used to maintain a destination oriented DAG in a packet radio network when the destination is static, is used in [32] to maintain a token oriented DAG with a dynamic destination. Like the algorithms of [7, 10, 29], each processor in this algorithm maintains a request queue containing the identifiers of neighboring processors from which it has received requests for the token.

The KRL algorithm maintains k tokens in the system as in [5, 31]. 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, our algorithm ensures that all non-token holding processors will always have a path to some token holding processor. In the KRL algorithm, each node dynamically chooses its lowest neighbor as its preferred link to a token holder (cf. [32]). 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.

We have improved the KRL algorithm by modifying it so that processors that receive the token or leave the CS and have no pending requests immediately send the token to some other neighbor. To keep the tokens from becoming localized in certain areas of the network, we require a token holder with no requests pending to send the token to a different neighbor than the one it last sent the token to when possible.

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 on the mobile nodes and network are:

1. the nodes have unique node identifiers in the range $0 \ldots n - 1$,

2. communication links are bidirectional and FIFO,

3. a link-level protocol ensures that each node is aware of its neighbors, i.e., the set of nodes with which it can currently directly communicate, by providing indications of link formations and failures, and

4. incipient link failures are detectable, providing reliable communication on a per-hop basis.

The only restriction we place on node failures is that not all current token holders fail during an execution. Also, partitions of the network are allowable, since the portion of the network that includes at least one token holder can continue running the algorithm with the subset of tokens that partition holds.
Each node has a $k$-mutual exclusion process, modeled as a state machine (see Figure 1), with the usual set of states, some of which are initial states, and a transition function. Each state at processor $i$ contains a local variable that holds the node identifier and a local variable that holds the identifiers of all nodes in direct wireless contact with node $i$, the current neighbors of $i$.

![Diagram of the system architecture](image)

**Figure 1: System architecture.**

A step of the $k$-mutual exclusion process at node $i$ is triggered by the occurrence of an input event. 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. Referring to Figure 1, the application $I/O$ events at the $k$-mutual exclusion process are:

1. Request\textsubscript{CS}$_i$: (input) request for access to CS.
2. Release\textsubscript{CS}$_i$: (input) release of CS.
3. Enter\textsubscript{CS}$_i$: (output) permission to enter CS.

The network $I/O$ events at the $k$-mutual exclusion process are:

1. \text{Recv}$_i(j,m)$: (input) message $m$ from node $j$ received at $i$.
2. \text{LinkUp}$_i(l)$: (input) link $l$ incident on $i$ has formed.
3. \text{LinkDown}$_i(l)$: (input) link $l$ incident on $i$ has failed.
4. \text{Send}$_i(j,m)$: (output) node $i$ sends message $m$ to $j$.
An execution is a sequence of the form $C_0, in_1, out_1, C_1, in_2, out_2, C_2, \ldots$, where the $C_k$'s are configurations, the $in_k$'s are input events, and the $out_k$'s are sets of output events. An execution must end in a configuration if it is finite. A positive real number is associated with each $in_i$, representing the time at which that event occurs. An execution must satisfy a number of additional conditions, which are listed now. The first set of conditions are basic “syntactic” ones.

- $C_0$ is an initial configuration.
- If $in_k$ occurs at node $i$, then $out_k$ and $i$'s state in $C_k$ are correct according to $i$'s transition function operating on $in_k$ and $i$'s state in $C_{k-1}$.
- The times assigned to the steps must be nondecreasing. If the execution is infinite, then the times must increase without bound. At most one step by each process can occur at a given time.

The next set of conditions require the application process to interact properly with the $k$-mutual exclusion process and to give up the CS in finite time.

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

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

- LinkUp$_i(l)$ occurs at time $t$ if and only if LinkUp$_j(l)$ occurs at time $t$, where $l$ joins $i$ and $j$. Furthermore, 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.

Finally, message delivery is considered. There must exist a one-to-one and onto correspondence between the occurrences of Send$_j(i, m)$ and 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 Send$_i(j, m)$ occurs at some time $t$, then the corresponding 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 $l$.

In every execution, the following must hold:

• If out$_k$ includes EnterCS$_i$, then the previous application event at node $i$ must be RequestCS$_i$. i.e., CS access is only given to requesting nodes.

We further require that any $k$-mutual exclusion algorithm satisfies the following properties:

1. $k$-mutual exclusion: At any time during the execution of the algorithm, at most $k$ processes can be in the CS.

2. no starvation: 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.

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

4 KRL Algorithm

In this section we first give a general overview of the operation of the KRL algorithm. Then we present examples of algorithm operation. Lastly, we present modifications to the algorithm that are intended to improve the overall fairness by ensuring that tokens are forwarded when not in use.

Each processor maintains a number of local data structures as part of the $k$-mutual exclusion process, including:

• 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 \((h_1, h_2, i)\) 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 higher neighbor. \(myHeight_i\) is included with every message sent by the \(k\)-mutual exclusion process on processor \(i\).

• **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, \(next = i\) if \(0 \leq i < k\), and \(next\) is an outgoing neighbor otherwise.

• **\(Q\)**: “Request queue”, containing identifiers of requesting neighbors and \(i\) if RequestCS\(_i\) was last application input event. 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, receivedLink[k] = true for all j ∈ Ni.

- formIn[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, formIn[j] = false for all j ∈ Ni.

- formHeight[j]: An array of tuples storing value of myHeight when new link to j first detected. Initially, formHeight[j] = myHeight for all j ∈ Ni.

4.1 Overview of algorithm

A DAG is maintained on the physical wireless links of the network throughout algorithm execution as the result of a three-tuple, or triple, of integers representing the “height” of the node, as in [13]. Links are considered to be directed from nodes with higher height toward nodes with lower height, based on lexicographic ordering of the triples. A link between two nodes is outgoing at the higher height node and incoming at the lower height node.

Node i’s height triple is included with every message sent by the k-mutual exclusion process on processor i, 0 ≤ i < n, where n is the number of participating processors. The three types of messages recognized by the algorithm are Request, Token, and LinkInfo. The purpose of each type of message should become clear in the discussion and examples below.

The algorithm maintains k tokens in the system. Initially the token holders are nodes 0...k − 1. We assume that k < n.

As described in the last section, the k-mutual exclusion algorithm is event-driven. When the application process on node i makes a request for the CS, i’s identifier is enqueued on its own request queue (Qi). Request messages received at node i from “higher” physical neighbors cause the k-mutual exclusion process at i to enqueue the identifiers of those neighbors on Qi in the order in which the Requests were received. Non-token holding node i sends a Request message to its lowest neighbor whenever an identifier is enqueued on an empty request queue at i. When i receives a Token message, it dequeues the top element on its request queue and either gives permission for its application process to enter the CS (if its own identifier was just dequeued) or sends a Token message to its neighboring node whose identifier was just dequeued.
Each token recipient \( i \) modifies the first two integers in its height triple if necessary each time it receives a \( \text{Token} \) message so that its height is lower than the height of the node that sent the \( \text{Token} \) message. This is not necessary when the node receiving the \( \text{Token} \) is already lower than the sender (which may be the case if the receiver holds or previously held a “lower” token.)

Non-token holding nodes must ensure that they have at least one “lower” neighbor at all times because requests for the token are always sent on outgoing paths. If a non-token holding node finds itself with no “lower” neighbor, it uses the \textit{partial reversal} technique of Gafni and Bertsekas [13] (found inside the \textit{RaiseHeight} procedure of Figure 4) to change the first two integers in its height triple, raising its height in relation to \( \geq 1 \) of its neighbors and creating at least 1 outgoing link. Each time a node raises its height, it sends \textit{LinkInfo} messages to all its neighbors. Request queue entries are deleted when the link to the requester fails or reverses. The reason requests are not lost as a result of these deletions is that a processor never deletes its own id from its request queue. Therefore, the request always has a chance to “repropagate” on a new route toward a token holder.

Token holders must ensure that they have at least one “higher” neighbor at all times, since \textit{Request} messages will not reach a token holder with only outgoing links. If a token holder finds itself with no “higher” neighbors, it uses the “reverse” of the Gafni and Bertsekas partial reversal technique [13] (found inside the \textit{LowerHeight} procedure of Figure 4) to change the first two integers in its height triple, lowering its height in relation to \( \geq 1 \) of its neighbors and creating at least 1 incoming link. Each time a node lowers its height (including when it receives a \textit{Token} message), it sends \textit{LinkInfo} messages on all its outgoing links.

\textbf{Pseudocode for KRL Algorithm}

The pseudocode for the KRL algorithm is presented in Figures 2 through 4. Each of the modules is assumed to be executed atomically.
When node \( i \) requests access to the CS:
1. \( \text{status} := \text{WAITING} \)
2. \( \text{Enqueue}(Q, i) \)
3. if \( \text{not tokenHolder} \)
4. if \( |Q| = 1 \) \( \text{ForwardRequest()} \)
5. else \( \text{GiveTokenToNext()} \)

When node \( i \) releases the CS:
1. if \( |Q| > 0 \) \( \text{GiveTokenToNext()} \)
2. \( \text{status} := \text{REMAIN} \)
3. if \( \text{myHeight} > \text{height}[k], \forall k \in N \)
4. \( \text{LowerHeight()} \)

---

**Figure 2:** Pseudocode triggered by input events from application process.

When \( \text{Request}(h) \) received at node \( i \) from node \( j \):
- \( h \) denotes \( j \)'s height when message was sent
1. if \( \text{receivedLL}[j] \)
2. \( \text{height}[j] := h \) // set \( i \)'s view of \( j \)'s height
3. if \( \text{myHeight} < \text{height}[j] \) \( \text{Enqueue}(Q, j) \)
4. if \( \text{tokenHolder} \)
5. if \( (|Q| > 0) \) and ((status = REMAIN) or \((\text{status} = CRITICAL) \) and \( \text{(numTokens > 1)} \))
6. \( \text{GiveTokenToNext()} \)
7. else // not tokenHolder
8. if \( \text{myHeight} < \text{height}[k], \forall k \in N \)
9. \( \text{RaiseHeight()} \)
10. else if \( (|Q = [j]) \) or \((|Q | > 0) \)
11. \( \text{ForwardRequest()} \) //reroute request

When \( \text{LinkInfo}(h) \) received at node \( i \) from node \( j \):
1. \( N := N \cup \{j\} \)
2. if \( (\text{forming}[j]) \) and \( \text{myHeight} \neq \text{formHeight}[j]) \)
3. \( \text{Send LinkInfo(myHeight) to } j \)
4. \( \text{forming}[j] := \text{false} \)
5. if \( \text{receivedLL}[j] \)
6. \( \text{Delete}(Q, j) \)

When failure of link to \( j \) detected at node \( i \):
1. \( N := N \cup \{j\} \)
2. \( \text{Delete}(Q, j) \)
3. \( \text{receivedLL}[j] := \text{true} \)
4. if \( \text{not tokenHolder} \)
5. if \( \text{myHeight} < \text{height}[k], \forall k \in N \)
6. \( \text{RaiseHeight()} \) //reroute request
7. else if \( (|Q| > 0) \) and \( \text{next} \neq N \)
8. \( \text{ForwardRequest()} \)
9. else if \( \text{myHeight} > \text{height}[k], \forall k \in N \)
10. \( \text{LowerHeight()} \)

When formation of link to \( j \) detected at node \( i \):
1. \( \text{Send LinkInfo(myHeight) to } j \)
2. \( \text{forming}[j] := \text{true} \)
3. \( \text{formHeight}[j] := \text{myHeight} \)

---

**Figure 3:** Pseudocode triggered by \( \text{Recv}(j, m) \) \( (m = \text{Request, Token, LinkInfo, LinkDown and LinkUp} \) network input events.

11
4.2 Example of static KRL Operation

An illustration of algorithm operation on a static network (in which links do not fail or form) is depicted in Figure 5. Snapshots of the system configuration during algorithm execution are shown, with time increasing from 5(a) to 5(f). In Figures 5, 6, 7, and 8, 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 solid arrows (local variable next) represent links over which either Token or Request messages have most recently been sent. Note that when a node holds a token, its next pointer is directed itself.

In Figure 5(a), nodes 2, 3, and 4 have requested access to the CS (note that nodes 2, 3, and 4 have enqueued themselves on Q₂, Q₃, and Q₄, respectively) and nodes 2 and 3 have sent Request messages to node 0, which enqueued them on Q₀ 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 Q₁ and Q₄). 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 Q₃). 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.
directed toward node 3 and both nodes 1 and 4 having next pointers directed toward node 4 in part (b).

Figure 5(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 released 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 all of its neighbors. This ensures that some future request may reach node 1.

In a static network, no node will have to raise its height. To see why, consider the operation of the algorithm in the absence of link changes. Nodes will lower their height (if necessary) when tokens are received or when they hold a token and have no incoming links. But this will cause no neighboring nodes to raise their height, since any affected non-token holding neighbors will gain an outgoing link.
4.3 Example of dynamic KRL algorithm operation

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 $next_i$. If the link to $next_i$ changes and $|Q_i| > 0$, node $i$ must reroute its request.

Identifier $j$ on the request queue at node $i$ is deleted if link $(i,j)$ fails or if $i$ raises its height so that the link to $j$ is outgoing. In the first case, node $j$ will be alerted with a network input event, and in the second case, $i$ will send a LinkInfo message to $j$. In either case, $j$ will be notified that its request for the CS will not be satisfied unless it sends a new Request message.

![Diagram](image)

**Figure 6:** Operation of KRL algorithm on dynamic network with 2 tokens.

Figure 6(a) shows the same snapshot of the system execution as is shown in Figure 5(a), with time increasing from 6(a) to 6(e). Figure 6(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.
4.4 KRL with token forwarding

This section describes a modification to the KRL algorithm designed to increase the circulation of tokens during execution. Within a connected component of the network, a token is idle at node \( i \) when there is a non-token holding processor \( j \) in its WAITING section at the same time \( i \) is in its REMAINDER section with \( |Q_i| = 0 \).

![Diagram of token problem in KRL algorithm]

Figure 7: Idle token problem in KRL algorithm.

Figure 7 shows why tokens are frequently idle in the KRL algorithm. The figure gives a snapshot of KRL execution in which there are 2 token holders, nodes 3 and 5. Nodes 4 and 6 have made requests and have sent Request messages to node 5. The application process at node 5 is currently in the CS, so node 5 has enqueued the identifiers of node 6 and node 4 on its request queue. The application process at node 3 has already released the CS and node 3 is in its REMAINDER section with an empty request queue. Therefore, node 3 holds an idle token. Node 3 will not send its token to any other node until it receives a Request message. Meanwhile, nodes 6 and 4 must wait their turns for the token being used by node 5.

We try to alleviate the idle token problem by having each token holder forward the token to other parts of the network in case no processor close to it needs access to the CS. The strategy we use is to mimic the action taken by processors when forwarding a request for a token, i.e., choose the “lowest” neighboring node and send the token to that neighbor. Choosing the lowest height neighbor results in the lowest number of link reversals because the lower the height of a neighbor, the fewer outgoing links that neighbor will need to reverse when it receives the token. Nodes keep track of which of their neighbors they have forwarded tokens to or received tokens from when their request queue is empty by marking the link “V” for “visited”.

Figure 8 illustrates this modification during an execution of the algorithm. In Figure 8(a), node 3 is a token holder but no neighbor of node 3 needs access to the CS. Figure 8(b) shows a snapshot of the algorithm execution after the application process on node 3 has released the CS, and the token has been forwarded through processors 0, 2, 1, and 3 to the left portion of the network. The V on each wireless link signifies that the link has been marked “visited” by both the node forwarding and the node receiving the
token. If node 3 receives the token at a later time, while it has an empty request queue, it will mark all its links as “unvisited” and start the forwarding process over.

**Pseudocode for KRL with token forwarding**

For every node, we add the following local data structure:

- $\text{visited}[j]$: boolean array indicating whether a token has been circulated to node $j$. Initially set to false for all $j \in N$.

For every node, we add and modify the modules listed in Figure 9. All other modules remain the same as specified in the last section.

---

**Figure 9:** Pseudocode modifications for token forwarding.

---
5 Correctness of KRL 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.

The full proof of no starvation for the KRL algorithm can be found in [33]. To save space, we will just give an overview of the argument in this paper. To prove the KRL algorithm ensures no starvation, we first show that, after link changes cease, eventually processors will stop raising their heights and the DAG will be token oriented. Then we show that any sequence of propagated requests, or “request chain”, beginning at any requesting processor will eventually include some token holder. Lastly, using a variant function argument, we show that a token will be delivered to every requesting node.

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

The token forwarding modification described in the last section to circulate idle tokens in the network will not violate the correctness of the algorithm. To see why, consider that there are a finite number of processors in the network and that the token cannot indefinitely “outrun” a request chain after link failures cease. Therefore, every request chain must eventually include some token holder and the proof of correctness holds.

6 Simulation Results

We simulated a 30 node system under various scenarios using an object-oriented discrete event simulator first developed and tested in [32]. 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 [3, 4, 6, 17, 20, 30].

In 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 $\lambda_{req}$. 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 $\lambda_{mob}$. 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 ($\lambda_{req}$), the degree of mobility ($\lambda_{mob}$), 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.

To find out how well our algorithm performs in comparison to another distributed $k$-mutual exclusion algorithm, we simulated the algorithm of Bulgannawar and Vaidya [5]. We chose this algorithm because it achieves lower delay and number of messages per CS entry than other static $k$-mutual exclusion algorithms. We simulated the Bulgannar and Vaidya algorithm (hereafter called BVR) as if it were running on top of an idealized ad hoc routing protocol that always found shortest paths between pairs of nodes. When routes changed during periods of mobility, we did not count the messages or time needed for route discovery. Only the time and number of messages needed for retracing active routes (routes in which messages were in transit at the time the route was disrupted) were charged to the BVR simulation. The underestimation of routing costs makes our simulation results more generally applicable because if our algorithm performs better than BVR in some simulated situation, then it would certainly perform better when all routing costs were actually charged to BVR.

Both the BVR and KRL algorithms start with nodes with identifiers ranging from 0 to $k - 1$ holding tokens. In KRL, we initially adjusted the height of each token holder to ensure that it had at least one incoming link. A connected graph whose initial edges were chosen at random with the desired number of links was generated, node heights and link directions were initialized, and then the algorithm and performance measurements were started. For BVR, we initially created $k$ logical trees of depth 1 on the network, rooted at each token holder. 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.

Figures 10 and 11 are plots of the average number of time units and the average number of messages per CS entry for the BVR, KRLF, and KRL simulations, respectively. These figures plot the time units and messages against values of $\lambda_{req}$ 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 1 for the high load value of $\lambda_{req}$ because at this rate each node would have a request pending almost all the time. The low load value of $\lambda_{req} = 10^{-3}$ 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). In
Figure 10: Load vs. time/CS entry for (a) zero, (b) low (1 link change every 500 time units), and (c) high (1 link change every 50 time units) mobility, $k = 3$ (BVR = Bulgannawar, Vaidya and router simulation, KRL = basic $k$-mutual exclusion algorithm, KRLF = KRL with token forwarding).

Figure 11: Load vs. messages/CS entry for (a) zero, (b) low (1 link change every 500 time units), and (c) high (1 link change every 50 time units) mobility, $k = 3$ (BVR = Bulgannawar, Vaidya and Router simulation, KRL = basic $k$-mutual exclusion algorithm, KRLF = KRL with token forwarding).

These figures, part (a) displays results when the graph is static, part (b) when $\lambda_{mob} = 50^{-2}$ (low mobility), and part (c) when $\lambda_{mob} = 50^{-1}$ (high mobility). Our choice for the value of the low mobility parameter corresponds to the situation where nodes remain stationary for up to a minute 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 tens of seconds between moves.

Figure 12 focuses on a fixed load and shows the time per CS entry (part (a)) and messages per CS
entry (part (b)) for executions of BVR, KRLF, and KRL where the mean time between requests is 10 time units for networks ranging from 44 links (10% connectivity) to 348 links (80% connectivity).

![Graphs](image)

Figure 12: Connectivity vs. (a) time units/CS entry and (b) messages/CS entry for zero, low (1 link change every 500 time units), and high (1 link change every 50 time units) mobility, \( k = 3 \), mean time between requests = 10 time units (BVR = Bulgamawar, Vaidya and router simulation, KRLF = KRL with token forwarding, KRL = basic \( k \)-mutual exclusion algorithm).

### 6.1 Comparison of KRL and KRLF

Figure 10 shows that KRLF (KRL algorithm with forwarding) results in executions with lower average time per CS entry. KRLF uses less than half the time per CS entry that KRL uses when the mean time between requests is 10 time units and nodes are static. The KRL algorithm creates logical partitions in the network, circulating the same token among the same nodes for the entire execution. The KRLF algorithm breaks these logical partitions by allowing token holders to send idle tokens to random neighbors. At this system load, it appears that this random “re-partitioning” of logical token routes allows for good circulation patterns and better overall token distribution.

Figure 11 shows that, for loads ranging from 0.1 to 1, the KRLF algorithm is comparable, in terms of number of message per CS entry, to the KRL algorithm. At the lowest loads, KRLF uses more messages than KRL due to the continuous circulation of tokens, demonstrating that the token forwarding strategy is wasteful when load is low. However, increasing token circulation in the network appears to have performance benefits, particularly at medium to high loads.

From Figure 12(a), we can see that, at connectivities ranging from 10% to 80% and a fixed load of
10 time units per request, the KRL algorithm performs better in terms of time per CS entry as mobility increases. The KRLF algorithm takes less time per CS entry than KRL and the performance is unchanged as mobility increases. This reflects the ability of the KRL and KRLF algorithms to maintain high levels of token usage even when the topology is very dynamic. By allowing a token holder to "forget" requests that arrive on links that subsequently fail, KRL is able to provide uninterrupted access to other neighbors of that token holder. However, there is a cost for the improvement in time per CS entry, because more messages are sent as mobility increases in KRL, as can be seen in Figure 12, part (b). These messages are sent as non-token holders lose their last outgoing links, raise their heights, and send LinkInfo messages to all their neighbors.

These results indicate that the KRLF algorithm has advantages, in terms of time per CS entry, over the KRL algorithm, particularly at mid-range system loads. From Figure 11, we can see that the token forwarding strategy is more costly, in terms of number of messages sent, only when the load is low or when nodes are static.

6.2 Comparison of BVR with KRL and KRLF

Figure 10 shows that the time per CS entry of the BVR simulation is lower at all system loads and mobility levels when the connectivity of the network is high than it is when connectivity is low. Figure 11 shows that the number of messages per CS entry for BVR is also lower in all cases when connectivity is high than it is when connectivity is low.

Figures 10 and 11 suggest that the BVR algorithm functions best at high connectivity. As mobility increases in parts (b) and (c), the time and number of messages per CS entry of the KRL and KRLF algorithms are nearly equal to, and in some cases, better than, the BVR simulation, even when connectivity is high. Also, Figure 10 shows that the KRLF algorithm can cut the wait time per CS entry roughly in half at mid-range system loads.

Figure 12(a) shows that, at a load of 10 time units per request, the KRL algorithm uses less time per CS entry than the BVR simulation at connectivities below 20% and the KRLF algorithm uses less time than BVR at connectivities below 50%. Part (b) of Figure 12 shows that when the mean time between requests is 10 time units, the KRL and KRLF algorithms send fewer messages per CS entry than BVR at all connectivities when nodes are static and when mobility is low. When mobility is high, the KRL and KRLF algorithms send fewer messages than BVR at connectivities below 70%.

The results in Figure 12 indicate that the performance of KRL and KRLF at a given load is less affected by the network connectivity than is the BVR simulation. The performance of the BVR simulation degrades as the network connectivity decreases. The relative insensitivity to connectivity exhibited by the KRL and KRLF algorithms may make these algorithms better suited than their static counterparts (such as BVR).
to the variable connectivity found in mobile ad hoc networks. Since high connectivity in mobile networks causes increased message collisions, an algorithm that relies on high connectivity, such as BVR does, may be less efficient than our topology sensitive algorithms in the ad hoc environment.

6.3 Experiments combining token forwarding and holding

This section presents the results obtained by a heuristic modification to the KRLF token forwarding strategy, a combination token forwarding and token holding algorithm we call KRLHF (KRL with “hold before forward”). During periods of low load, processors hold idle tokens for some specified time period $\theta$ prior to forwarding the token.

In our simulation, we set $\theta = 200$ time units. The results are shown in Figures 13 and 14.

![Figure 13: Load vs. time units/CS entry for zero, low (1 link change every 500 time units), and high (1 link change every 50 time units) mobility, $k = 3$, connectivity = (a) 20\% and (b) 80\% (KRLF = KRL with token forwarding, KRLHF = KRL with “hold before forward” heuristic).](image)

The most significant improvement shown in Figure 14 can be seen in part (b), where the number of messages per CS entry used by KRLHF is roughly half that of KRLF at the lowest system load when nodes are static. This figure also shows that the number of messages per CS entry used by KRLHF is substantially lower than KRLF at the lowest load when nodes are mobile. In Figure 14(a), the number of messages used by KRLHF per CS entry is roughly half that of KRLF at the lowest system load when nodes are static and mobility is low. Both Figures 13 and 14 show that when the system load is high, the KRLHF algorithm uses about the same number of messages and time per CS entry as the KRLF algorithm. This indicates that token forwarding is not as beneficial at higher loads, when nodes will more likely have
Figure 14: Load vs. messages/CS entry for zero, low (1 link change every 50 time units), and high (1 link change every 50 time units) mobility, $k = 3$, connectivity = (a) 20% and (b) 80% (KRLF = KRL with token forwarding, KRLHF = KRL with “hold before forward” heuristic).

requests enqueued whenever the CS is released. Figures 13 and 14 also show that the decrease in message complexity for KRLHF has little effect on waiting time.

7 Conclusion and Discussion

We have presented a topology sensitive $k$-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 also presented a token forwarding technique to improve token circulation.

Through simulation, we showed that at mid-range loads, the token forwarding technique does improve the time per CS entry without using more messages than the basic KRL algorithm when nodes are mobile. We presented a comparison of our algorithms to a distributed $k$-mutual exclusion algorithm designed for a completely connected network [5]. Our results suggest that our algorithms perform better in many cases than the static algorithm and ad hoc router combination.

We are continuing to experiment with heuristic modifications to the token forwarding strategy. A “self-adjusting” version of KRLHF would allow processors to dynamically adjust the amount of time they hold an idle token before forwarding, based on their local view of the load on the system. Our simulation results suggest that this strategy will not degrade performance even if the node has an incorrect view of the system load.
References


