

A 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,g0c7670,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 and to allow the tokens to circulate more evenly among participating processors.

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 KRL algorithm presented here includes new techniques to promote fair access to the CS and to maintain the DAG with multiple tokens. The algorithm ensures that all requesting processors eventually gain access to the CS once the network stabilizes and communication links are reestablished. A 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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POMC '01 Newport, Rhode Island USA
Copyright 2001 ACM 1-58113-397-9/01/08 ...\$5.00.

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 en-

sure 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 $0 \dots n - 1$,
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 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 .

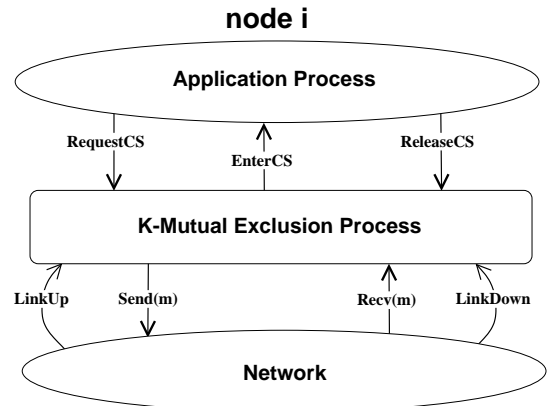


Figure 1: System architecture.

A step of the 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 mutual exclusion process are:

1. RequestCS _{i} : (input) request for access to CS.
2. ReleaseCS _{i} : (input) release of CS.
3. EnterCS _{i} : (output) permission to enter CS.

The *network I/O events* at the 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 .

We 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. The complete list of local data structures at each node and the pseudocode for the algorithm can be found in the appendix.

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 three tuples. A link between two nodes is *outgoing* at the higher height node and *incoming* at the lower height node. A total ordering on the height of nodes in the network is ensured because the last integer in the triple is the unique identifier of the node. For example, if the height at node 1 = (2, 3, 1) and the height at node 2 = (2, 2, 2), then the link between these nodes would be directed from node 1 to node 2. Initially at node 0, height = (0, 0, 0) and, for all $i \neq 0$, i ’s height is initialized so that the directed links form a DAG in which every non-token holder has a directed path to some token holder and every token holder has at least one incoming link.

Node i ’s height triple is included with every message sent by the mutual exclusion process on processor i , $0 \leq i < n$, 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 $0 \dots k - 1$. We assume that $k < n$.

As described in the last section, the 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 (Q_i). *Request* messages received at node i from “higher” physical neighbors causes the mutual exclusion process at i to enqueue the identifiers of those neighbors on Q_i 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 *Token* message so that its height is lower than the height of the node that sent the *Token* message. This is not necessary when the node receiving the *Token* message has lower values in the first two integers of its height triple than the sending node.

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 *partial reversal* technique of Gafni and Bertsekas [13] to change the first two integers in its height triple, raising its height in relation to ≥ 1 of its neighbors and creating at least 1 outgoing link. Each time a node raises its height, it sends *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 so that *Request* messages can be delivered to them. If a token holder finds itself with no “higher” neighbors, it uses the “reverse” of the Gafni and Bertsekas partial reversal technique [13] to change the first two integers in its height triple, lowering its height in relation to ≥ 1 of its neighbors and creating at least 1 incoming link. Each time a node lowers its height (including when it receives a *Token* message), it sends *LinkInfo* messages on all its outgoing links.

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 2. Snapshots of the system configuration during algorithm execution are shown, with time increasing from 2(a) to 2(f). In Figures 2, 3, 4, and 5, 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 towards itself.

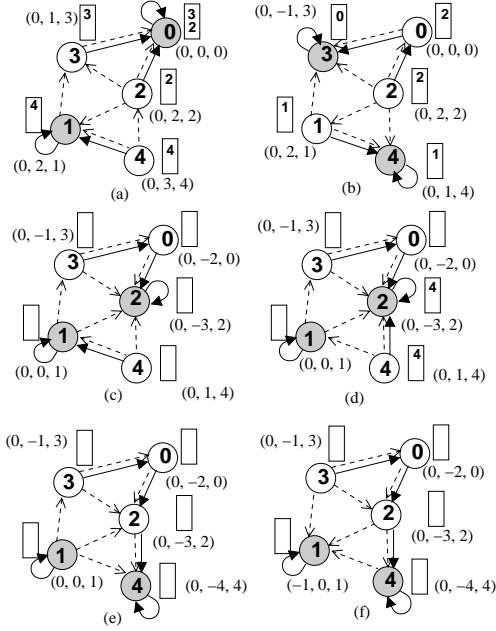


Figure 2: Operation of KRL algorithm on static network with 2 tokens.

In Figure 2(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 directed toward node 3 and both nodes 1 and 4 having *next* pointers directed toward node 4 in part (b).

Figure 2(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 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.

Figure 3(a) shows the same snapshot of the system execution as is shown in Figure 2(a), with time increasing from 3(a) to 3(e). Figure 3(b) depicts the system 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

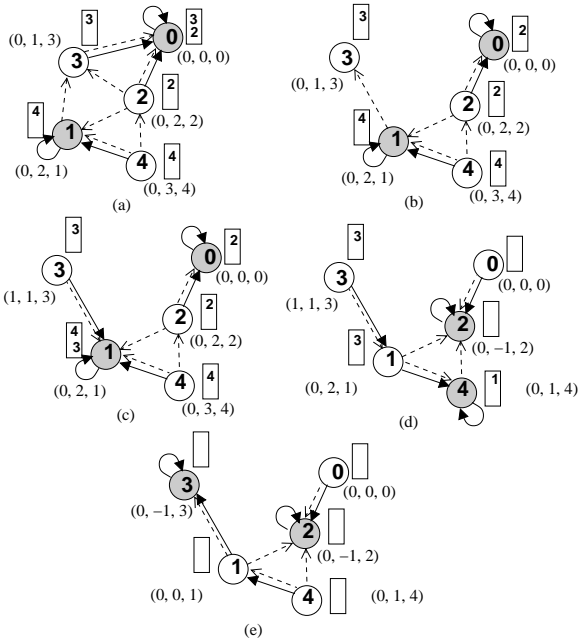


Figure 3: Operation of KRL algorithm on dynamic network with 2 tokens.

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 $|Q_i| = 0$.

Figure 4 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.

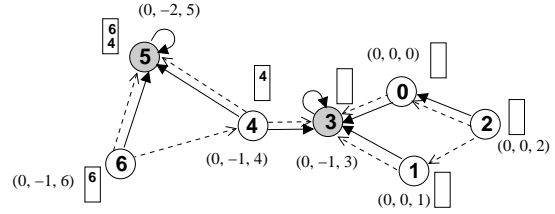


Figure 4: Idle token problem in KRL algorithm.

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 as “visited”.

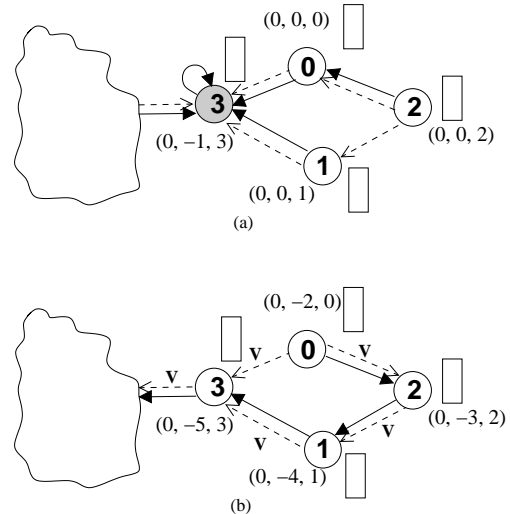


Figure 5: Operation of KRL algorithm with token forwarding.

Figure 5 illustrates this modification during an execution of the algorithm. In Figure 5(a), node 3 is a token holder but no neighbor of node 3 needs access to the CS. Figure 5(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.

The pseudocode detailing the modifications to the KRL algorithm can be found in the appendix.

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. Essentially, this proof is identical to the proof of correctness in [32].

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. 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 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 KRL simulation starts with nodes with identifiers ranging from 0 to $k - 1$ holding tokens. 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. 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.

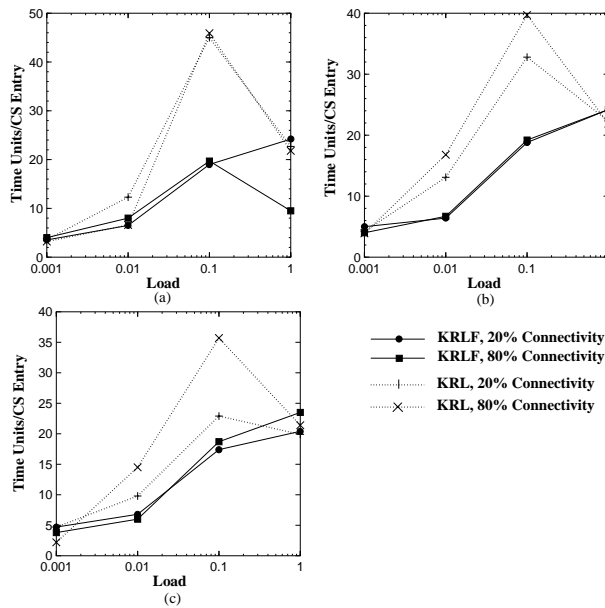


Figure 6: Load vs. time/CS entry for (a) zero, (b) low (1 link change every 500 time units), and (c) high (1 link change every 50 time units) mobility, $k = 3$ (KRL = basic k -mutual exclusion algorithm, KRLF = KRL with token forwarding).

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 1 for 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 $\lambda_{req} = 10^{-3}$ 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).

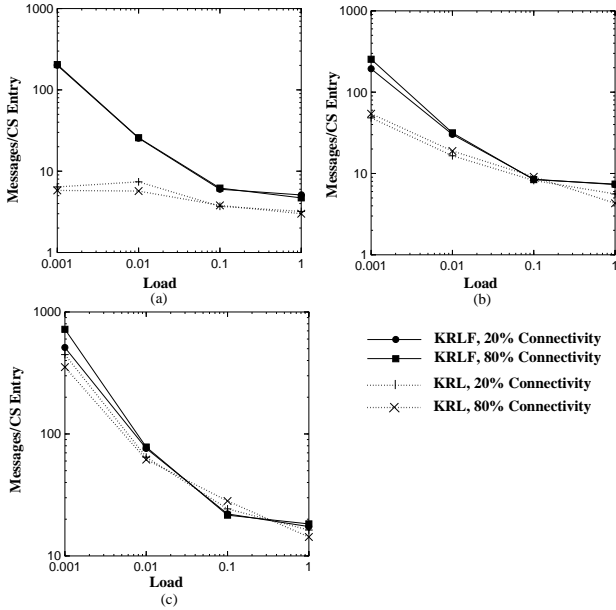


Figure 7: Load vs. messages/CS entry for (a) zero, (b) low (1 link change every 500 time units), and (c) high (1 link change every 50 time units) mobility, $k = 3$ (KRL = basic k -mutual exclusion algorithm, KRLF = KRL with token forwarding).

In Figures 6 and 7, 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.

Figures 6 and 7 show that KRLF (KRL algorithm with forwarding) results in executions with lower average time per CS entry, using less than half the time per CS entry as KRL uses when the mean time between requests is 10 time units. At this load, the KRLF algorithm actually uses *fewer* messages per CS entry when nodes are mobile, as can be seen in Figure 7, parts (b) and (c). Figure 7 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 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.

Figures 8 and 9 focus on a fixed load and show the time per CS entry and messages per CS entry in an execution

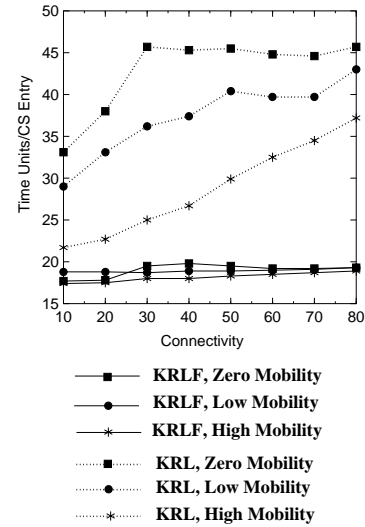


Figure 8: Connectivity vs. time units/CS entry for zero, low (1 link change every 500 time units), and high (1 link change every 50 time units) mobility, $k = 3$, mean time between requests = 10 time units (KRL = basic k -mutual exclusion algorithm, KRLF = KRL with token forwarding).

where the mean time between requests is 10 time units for networks ranging from 44 links (10% connectivity) to 348 links (80% connectivity).

Figure 8 shows that at connectivities ranging from 10% to 80%, the KRLF algorithm performs *better* in terms of time per CS entry as mobility increases. 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 9.

From the comparisons of the basic KRL algorithm to that of KRLF in Figure 8, we can see that at this fixed load, the KRLF version takes less than half the time used by the KRL algorithm per CS entry. Also, the KRLF algorithm is not sensitive to network connectivity at this load. Figure 9 shows that the KRLF algorithm also uses fewer messages per CS entry than the KRL algorithm at the highest mobility and at a load of 0.1.

From these results, it appears the KRLF algorithm has advantages over the KRL algorithm, particularly at mid-range system loads. The token forwarding strategy is more costly, in terms of number of messages sent, only at the lowest system loads.

7. CONCLUSION AND DISCUSSION

We have presented a topology sensitive k -mutual exclusion algorithm for mobile ad hoc networks. We directed the reader to the full proof 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

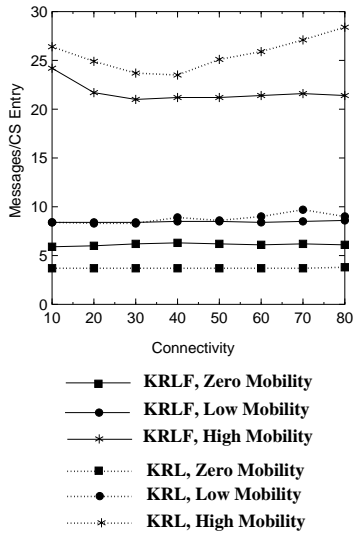


Figure 9: Connectivity vs. messages/CS entry for zero, low (1 link change every 500 time units), and high (1 link change every 50 time units) mobility, $k = 3$, mean time between requests = 10 time units (KRL = basic k -mutual exclusion algorithm, KRLF = KRL with token forwarding).

failures cease. We also have developed a token forwarding technique to improve the fairness of the algorithm.

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 are working on heuristic modifications to the token forwarding strategy, in which a processor adjusts the number of hops a token is forwarded based on its view of the load on the system. In this scheme, if a processor perceives that there is a low load on the system, it will not forward the token. We also plan to compare the performance of the KRLF algorithm to a “static” distributed k -mutual exclusion algorithm running on top of an ad hoc routing protocol.

8. REFERENCES

- [1] Y. Afek, E. Gafni, and A. Rosen. The slide mechanism with applications in dynamic networks. In *Proc. of 11th Annual Symp. on Prin. of Dist. Computing*, pages 35–46, 1992.
- [2] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial end to end communication. In *Proc. of 30th Annual Symp. on Found. of Comp. Sci.*, pages 358–363, 1989.
- [3] S. Basagni, I. Chlamtac, and V. R. Syrotiuk, “A Distance Routing Effect Algorithm for Mobility (DREAM),” *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM ’98)*, pp. 76–84, 1998.
- [4] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, and J. Jetcheva, “A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols,” *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM ’98)*, pp. 85–97, 1998.
- [5] 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.
- [6] R. Casteneda and S. R. Das, “Query Localization Techniques for On-Demand Routing Protocols in Ad Hoc Networks,” *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM ’99)*, pp. 186–194, 1999.
- [7] Y. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. of 9th IEEE Symp. on Reliable Dist. Systems*, pages 146–154, 1990.
- [8] C. Chiang and M. Gerla. Routing and multicast in multihop, mobile wireless networks. In *Proc. of ICUPC ’97*, pages 546–551, 1997.
- [9] M. S. Corson and A. Ephremides. A distributed routing algorithm for mobile wireless networks. *ACM J. Wireless Networks*, 1(1):61–81, 1997.
- [10] D. M. Dhamdhere and S. S. Kulkarni. A token based k -resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50:151–157, 1994.
- [11] R. Dube, C. D. Rais, K. Wang, and S. K. Tripathi. Signal stability based adaptive routing (SSA) for ad-hoc mobile networks. *IEEE Personal Communications*, pages 36–45, Feb. 1997.
- [12] A. Ephremides and T. V. Truong. Scheduling broadcasts in multihop radio networks. *IEEE Trans. on Communications*, 38(4):456–460, 1990.
- [13] E. Gafni and D. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, C-29(1):11–18, 1981.
- [14] M. Gerla and T.-C. Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, pages 255–265, 1995.
- [15] K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, and R. B. Tan, “Fundamental Control Algorithms in Mobile Networks,” *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pp. 251–260, 1999.
- [16] 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.
- [17] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark, “Scenario-Based Performance Analysis of Routing Protocols for Mobile Ad-Hoc Networks,” *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM ’99)*, pp. 195–206, 1999.
- [18] 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.

- [19] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proc. of 15th Annual Symp. on Prin. of Dist. Computing*, pages 68–76, 1996.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] N. Malpani, J. L. Welch, and N. H. Vaidya, “Leader Election Algorithms for Mobile Ad Hoc Networks,” *Proc. Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pp. 96–103, 2000.
- [24] E. Pagani and G. P. Rossi. Reliable broadcast in mobile multihop packet networks. In *Proc. of ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '97)*, pages 34–42, 1997.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(1989):189–193, 1989.
- [29] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
- [30] E. M. Royer and C. E. Perkins, “Multicast Operation of the Ad-Hoc On-Demand Vector Routing Protocol,” *Proc. ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '99)*, pp. 207–218, 1999.
- [31] 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.
- [32] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. Accepted to *ACM and Baltzer Wireless Networks journal, special issue on DialM papers*, 2001.
- [33] J. E. Walter. A k-mutual exclusion algorithm for ad hoc mobile networks. Technical report 00-022, Texas

A&M University, 2000,

<http://www.cs.tamu.edu/people/jennyw/kmutex.ps.gz>.

APPENDIX: KRL ALGORITHM

Each processor maintains a number of local data structures as part of the 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 a the 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, $receivedLI[k] = true$ 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 $j \in N_i$.

Pseudocode for KRL Algorithm

Each of the following modules is assumed to be executed atomically.

When node i requests access to the CS:

1. $status := WAITING$
2. $Enqueue(Q, i)$
3. if (not $tokenHolder$)
4. if ($|Q| = 1$) $ForwardRequest()$
5. else $GiveTokenToNext()$

When node i releases the CS:

1. if ($|Q| > 0$) $GiveTokenToNext()$
2. $status := REMAINDER$
3. if ($myHeight > height[k], \forall k \in N$)
4. $LowerHeight()$

When $Request(h)$ received at node i from node j :

- // h denotes j 's height when message was sent
1. if ($receivedLI[j]$)
 2. $height[j] := h$ // set i 's view of j 's height
 3. if ($myHeight < height[j]$) $Enqueue(Q, j)$
 4. if ($tokenHolder$)
 5. if ($(|Q| > 0)$ and ($(status = REMAINDER)$ or ($status = CRITICAL$ and ($numTokens > 1$))))
 6. $GiveTokenToNext()$
 7. else // not tokenholder
 8. if ($myHeight < height[k], \forall k \in N$)
 9. $RaiseHeight()$
 10. else if ($(Q = [j])$ or ($(|Q| > 0)$ and ($myHeight < height[next]$)))
 11. $ForwardRequest()$ //reroute request

When $Token(h)$ received at node i from node j :

- // h denotes j 's height when message was sent
1. $tokenHolder := true$
 2. $numTokens++$
 3. $height[j] := h$
 4. if ($myHeight > h$)
 5. $Send LinkInfo(h.h1, h.h2 - 1, i)$
to all outgoing $k \in N$ except j
 6. $myHeight.h1 := h.h1$
 7. $myHeight.h2 := h.h2 - 1$ // lower my height
 8. $Send LinkInfo(h.h1, h.h2 - 1, i)$ to j
 9. if ($|Q| > 0$) $GiveTokenToNext()$
 10. else $next := i$

When $LinkInfo(h)$ received at node i from node j :

- // h denotes j 's height when message was sent
1. $N := N \cup \{j\}$
 2. if ($(forming[j])$ and ($myHeight \neq formHeight[j]$))
 3. $Send LinkInfo(myHeight)$ to j
 4. $forming[j] := false$
 5. if ($receivedLI[j]$) $height[j] := h$
 6. else if ($height[j] = h$) $receivedLI[j] := true$
 7. if ($myHeight > height[j]$) $Delete(Q, j)$
 8. if ($tokenHolder$)
 9. if ($myHeight > height[k], \forall k \in N$)
 10. $LowerHeight()$
 11. if ($(myHeight < height[k], \forall k \in N)$ and (not $tokenHolder$))
 12. $RaiseHeight()$
 13. else if ($(|Q| > 0)$ and ($myHeight < height[next]$))
 14. $ForwardRequest()$ // 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 $tokenHolder$)
5. if ($myHeight < height[k], \forall k \in N$)

6. $RaiseHeight()$ // reroute request
7. else if ($(|Q| > 0)$ and ($next \notin N$))
8. $ForwardRequest()$
9. else if ($myHeight > height[k], \forall k \in N$)
10. $LowerHeight()$

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

Procedure $GiveTokenToNext()$:

1. $next := Dequeue(Q)$
2. if ($next \neq i$)
3. $numTokens--$
4. if ($numTokens = 0$)
5. $tokenHolder := false$
6. $height[next] := (myHeight.h1, myHeight.h2 - 1, next)$
7. $receivedLI[next] := false$
8. $Send Token(myHeight)$ to $next$
9. if ($(numTokens = 0)$ and ($|Q| > 0$))
10. $Send Request(myHeight)$ to $next$
11. else // $next = i$
12. $status := CRITICAL$
13. Enter CS

Procedure $RaiseHeight()$:

1. $myHeight.h1 := 1 + \min_{k \in N} \{height[k].h1\}$
2. $S := \{l \in N : height[l].h1 = myHeight.h1\}$
3. if ($S \neq \emptyset$) $myHeight.h2 := \min_{l \in S} \{height[l].h2\} - 1$
4. $Send LinkInfo(myHeight)$ to all $k \in N$
5. for (all $k \in N$ such that $myHeight > height[k]$) do
6. $Delete(Q, k)$
7. if ($|Q| > 0$) $ForwardRequest()$

Procedure $LowerHeight()$:

1. $myHeight.h1 := \max_{k \in N} \{height[k].h1\} - 1$
2. $S := \{l \in N : height[l].h1 = myHeight.h1\}$
3. if ($S \neq \emptyset$) $myHeight.h2 := \max_{l \in S} \{height[l].h2\} + 1$
4. $Send LinkInfo(myHeight)$ to all incoming $k \in N$

Pseudocode for KRL with token forwarding

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

- $visited[j]$: 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 below. All other modules remain the same.

When node i releases the CS:

1. if ($|Q| > 0$) $GiveTokenToNext()$
2. else
3. $PickLowest \& ForwardToken()$
6. $status := REMAINDER$

When $Token(h)$ received at node i from node j :

- // h denotes j 's height when message was sent
1. $visited[j] := true$
 2. $tokenHolder := true$
 3. $numTokens++$
 4. $height[j] := h$
 5. if ($myHeight > h$)
 6. $Send LinkInfo(h.h1, h.h2 - 1, i)$
to all outgoing $k \in N$ except j
 7. $myHeight.h1 := h.h1$
 8. $myHeight.h2 := h.h2 - 1$ // lower my height
 9. $Send LinkInfo(h.h1, h.h2 - 1, i)$ to j
 10. if ($|Q| > 0$) $GiveTokenToNext()$
 11. else
 12. $PickLowest \& ForwardToken()$

When formation of link to j detected at node i :

1. Send *LinkInfo*(*myHeight*) to j
2. *forming*[j] := true
3. *formHeight*[j] := *myHeight*
4. *visited*[j] := false

Procedure *PickLowest@ForwardToken*():

1. if (*visited*[j] = true $\forall j \in N$)
2. *visited*[j] := false $\forall j \in N$
3. *next* := $l \in N : ((\text{height}[l] \leq \text{height}[j])$
 and (*visited*[j] = false) $\forall j \in N$)
4. *visited*[*next*] := true
5. *numTokens*--
6. if (*numTokens* = 0)
7. *tokenHolder* := false
8. *height*[*next*] := (*myHeight.h1*, *myHeight.h2-1*, *next*)
9. *receivedLI*[*next*] := false
10. Send *Token*(*myHeight*) to *next*