RESEARCH ARTICLE OPEN ACCESS

Structuring Distributed Algorithms for Mobile Hosts

R.Karthikeyan[1], Jayaprakash S[2] , Dharmalingam P[3]

[1] (Asst.Prof, Dept of MCA, Gnanamani college of Technology, Namakkal, INDIA)
[2](P.G.Scholar, Dept of MCA, Gnanamani college of Technology, Namakkal, INDIA)
[3](P.G.Scholar, Dept of MCA, Gnanamani college of Technology, Namakkal, INDIA)

Abstract:

Mobile Ad-hoc network is a self-configuring network of mobile routers connected by wireless links. This union forms a random topology. Mutual Exclusion in distributed mobile ad-hoc network ensures that only one process can access shared resources at a time. If at that time, other process requests for those shared resources, then the requesting process has to wait until the resources have been released. For mobility management, we present a algorithm which changes its communication according to the topology changes. In this algorithm we shows that the nodes are make communicate only with their current neighbors which yields more performance to adapt the mobility.

Keywords — **mobile adhoc networks, Distributed Mutual Exclusion, token based algorithms, dynamic nodes, mobility.**

1. Introduction

In the wireless communication, mobile *ad hoc* network is a network wherein a pair of nodes communicates by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes. Direct communication is possible only between pairs of nodes that lie within one another's transmission radius. Wireless link "failures" occurs when previously communicating nodes move such that they are no longer within transmission range of each other. Likewise, wireless link "formation" occurs when nodes that were too far separated to communicate move such that they are within transmission range of each other. Characteristics that distinguish ad hoc networks from existing distributed networks include frequent and unpredictable topology changes and highly variable message delays. These characteristics make ad hoc networks challenging environments in which to implement distributed algorithms.

The mutual exclusion problem involves a group of processes, each of which intermittently requires access to a resource or a piece of code called the *critical section* (CS). At most one process may be in the CS at any given time. Providing shared access to resources through mutual exclusion is a fundamental problem in computer science, and is worth considering for the ad hoc environment, where stripped down mobile nodes may need to share resources. Distributed mutual exclusion algorithms that rely on the maintenance of a logical structure to provide order and efficiency may be inefficient when run in a mobile environment, where the topology can potentially change with every node movement. We present an algorithm which dynamically modifying the logical structure to adapt to the changing physical topology in the ad hoc environment.

<table>
<tr><td colspan="2">User Applications</td></tr>
<tr><td>Distributed Primitives</td><td>Routing Protocol</td></tr>
<tr><td colspan="2">Ad Hoc Network</td></tr>
</table>

Existing distributed algorithms will run correctly on top of ad hoc routing protocols, since these protocols are designed to hide the dynamic nature of the network topology from higher layers in the protocol stack. Routing algorithms on ad hoc networks provide the ability to send messages from any node to any other node. However, our contention is that efficiency can be gained by developing a core set of distributed algorithms, or primitives, that are aware of the underlying mobility in the network, as shown in figure. In this paper, we present a *mobility aware* distributed mutual exclusion algorithm to illustrate the layering approach in figure. Distributed mutual exclusion algorithms that rely on the maintenance of a logical structure to provide order and efficiency may be inefficient when run in a mobile environment, where the topology can potentially change with every node movement. Badrinath et al. [3] solve this problem on cellular mobile networks, where the bulk of the

computation can be run on wired portions of the network.

We present a mutual exclusion algorithm that induces a logical directed acyclic graph (DAG) on the network, dynamically modifying the logical structure to adapt to the changing physical topology in the ad hoc environment.

2. Related Work

Token based mutual exclusion algorithms provide access to the CS through the maintenance of a single token that cannot simultaneously be present at more than one node in the system. Requests for CS entry are typically directed to whichever node is the current token holder. Raymond [1] introduced a token based mutual exclusion algorithm in which requests are sent, over a static spanning tree of the network, toward the token holder; this algorithm is resilient to non-adjacent node crashes and recoveries, but is not resilient to link failures. Chang et al. [3] extend Raymond's algorithm by imposing a logical direction on a sufficient number of links to induce a *token oriented DAG* in which, for every node *i*, there exists a directed path originating at *i* and terminating at the token holder. Allowing request messages to be sent over all links of the DAG provides resilience to link and site failures. However, this algorithm does not consider link recovery, an essential feature in a system of mobile nodes. Dhamdhere and Kulkarni [5] show that the algorithm of [3] can suffer from deadlock and solve this problem by assigning a dynamically changing sequence number to each node, forming a total ordering of nodes in the system. The token holder always has the highest sequence number, and, by defining links to point from a node with lower to higher sequence number, a token oriented DAG is maintained. Due to link failures, a node *i* that want to send a request for the token may find itself with no outgoing links to the token holder. In this situation, *i* flood the network with messages to build a temporary spanning tree. Once the token holder becomes part of such a spanning tree, the token is passed directly to node *i* along the tree, bypassing other requests. Since priority is given to nodes that lose a path to the token holder, it seems likely that other requesting nodes could be starved as long as link failures continue. Also, flooding in response to link failures and storing messages for delivery after link recovery make this algorithm ill-suited to the highly dynamic ad hoc environment. Our token based algorithm combines ideas from several papers. The partial reversal technique from [4], used to maintain a *destination oriented* DAG in a packet radio network when the destination is static, is used in our algorithm to maintain a token oriented DAG with a dynamic destination. Like the algorithms of [3,5,1], each node in our algorithm maintains a request queue containing the identifiers of neighboring nodes from which it has received requests for the token. Like [5], our algorithm totally orders nodes. The lowest node is always the current token holder, making it a "sink" toward which all requests are sent. Our algorithm also includes some new features. Each node dynamically chooses its lowest neighbor as its preferred link to the token holder. Nodes sense link changes to immediate neighbors and reroute requests based on the status of the previous preferred link to the token holder and the current contents of the local request queue. All requests reaching the token holder are treated symmetrically, so that requests are continually serviced while the DAG is being re-oriented and blocked requests are being rerouted.

3. 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:

- ✓ the nodes have unique node identifiers,
- ✓ node failures do not occur,
- ✓ communication links are bidirectional and FIFO,
- ✓ a link-level protocol ensures that each node is aware of the set of nodes with which it can currently directly communicate by providing indications of link formations and failures,
- ✓ Incipient link failures are detectable, providing reliable communication on a per-hop basis, and
- ✓ Partitions of the network do not occur.

4. Reverse Link (RL) Mutual Exclusion Algorithm

In this section we first present the data structures maintained at each node in the system, followed by an overview of the algorithm, the algorithm pseudocode, and examples of algorithm operation. Throughout this section, data structures are described for node i, $0 \leq i \leq n-1$. Subscripts on data structures to indicate the node are only included when needed.

➢ Data structures

• **status**: indicates whether node is in the WAITING, CRITICAL, or REMAINDER section. Initially, *status* = REMAINDER.

• **N:** the set of all nodes in direct wireless contact with node *i*. Initially, *N* contains all of node *i*'s neighbors.

• **myHeight:** a three-tuple (*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. For instance, 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 $i \neq 0$, *myHeighti* is initialized so that the directed links form a DAG in which every node has a directed path to node 0.

• **height[j]:** an array of tuples representing node *i*'s view of *myHeightj* for all *j Ni* . Initially, *height*[*j*] = *myHeightj*, for all *j Ni* . In node *i*'s viewpoint, if *j N*, then the link between *i* and *j* is *incoming* to node *i* if *height*[*j*] > *myHeight*, and *outgoing* from node *i* if *height*[*j*] < *myHeight*.

• **tokenHolder:** flag set to true if node holds token and set to false otherwise. Initially,
tokenHolder = true if *i* = 0, and *tokenHolder* = false otherwise.

• *next*: when node *i* holds the token, *next* = *i*, otherwise *next* is the node on an outgoing link.
Initially, *next* = 0 if *i* = 0, and *next* is an outgoing neighbor otherwise.

• ***Q:*** queue containing identifiers of requesting neighbors. Operations on *Q* include Enqueue( ), which enqueues an item only if it is not already on *Q*, Dequeue( ) with the usual FIFO semantics, and Delete( ), which removes a specified item from *Q*, regardless of its location. Initially, *Q* = .

• **receivedLI[j]:** Boolean array indicating whether *Link-Info* message has been received from node *j*, to which a *Token* message was recently sent. Any height information received at node *i* from a node *j* for which *receivedLI*[*j*] is false will not be recorded in *height*[*j*]. Initially, *receivedLIi*[*j*] = true for all *j Ni* .
when node i requests access to the CS

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

when node i releases the CS

1. if (|Q| > 0) GiveTokenToNext()
2. status := REMAINDER

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, *formingi*[*j*] = false for all *j Ni* .
formHeight[*j*]: an array of tuples storing value of *myHeight* when new link to *j* first detected. Initially, *formHeighti*[*j*] = *myHeighti* for all *j Ni* .

➢ **Overview of the RL algorithm**

The mutual exclusion algorithm is event-driven. An event at a node *i* consists of receiving a message from another node $j \neq i$, or an indication of link failure or formation from the link layer, or an input from the application on node *i* to request or release the CS. Each message sent includes the current value of *myHeight* at the sender. Modules are assumed to be executed atomically.

✓ ***Requesting and releasing the CS:*** When node *i* requests access to the CS, it enqueues its own identifier on *Q* and sets *status* to WAITING. If node *i* does not currently hold the token and *i* has a single element on its queue, it calls *ForwardRequest*( ) to send a *Request* message. If node *i* does hold the token, *i* can set *status* to CRITICAL and enter the CS, since it will be at the head of *Q*. When node *i* releases the CS, it calls *GiveTokenToNext*( ) to send a *Token* message if *Q* is non-empty, and sets *status* to REMAINDER.

✓ ***Request messages:*** When a *Request* message sent by a neighboring node *j* is received at node *i*, *i* ignores the *Request* if *receivedLI*[*j*] is false. Otherwise, *i* changes *height*[*j*], and enqueues *j* on *Q* if the link between *i* and *j* is incoming at *i*. If *Q* is non-empty, and *status* = REMAINDER, *i* calls *GiveTokenToNext*( ), provided *i* holds the token. Non-token holding node *i* calls *RaiseHeight*( ) if the link to *j* is now incoming and *i* has no outgoing links or *i* calls *ForwardRequest*( ) if *Q* = [*j*] or if *Q* is non-empty and the link to *next* has reversed.

✓ [illegible] *LinkInfo* message is received at node *i* from node *j* , *j* 's height is saved in *height*[*j*]. If *receivedLI*[*j*] is false, *i* checks if the height of *j* in the message is what it was when *i* sent the *Token* message to *j*. If so, *i* sets *receivedLI*[*j*] to true. If *forming*[*j*] is true, the current value of *myHeight* is compared to the value of *myHeight* when the link to *j* was first detected, *formHeight*[*j*]. If *myHeight* and *formHeight*[*j*] are different, then a *LinkInfo* message is sent to *j* . Identifier *j* is added to *N* and *forming*[*j*] is set to false. If *j* is an element of *Q* and *j* is an outgoing link, then *j* is deleted from *Q*. If node *i* has no outgoing links and is not the token holder, *i* calls *RaiseHeight*( ) so that an outgoing link will be formed. Otherwise, if *Q* is non-empty, and the link to *next* has reversed, *i* calls *ForwardRequest*( ) since it must send another *Request* for the token.

✓ ***Link formation:*** When node *i* detects a new link to node *j* , *i* sends a *LinkInfo* message to *j* with *myHeight*,

sets *forming*[*j*] to true, and sets *formHeight*[*j*] = *myHeight*.

- ✓ ***Procedure ForwardRequest:*** Selects node *i*'s lowest height neighbor to be *next*. Sends a *Request* message to *next*.

- ✓ ***Procedure GiveTokenToNext:*** Node *i* dequeues the first node on *Q* and sets *next* equal to this value. If *next* = *i*, *i* enters the CS. If *next* ≠ *i*, *i* lowers *height*[*next*] to (*myHeight.h*1, *myHeight.h*2 − 1, *next*), so any incoming *Request* messages will be sent to *next*, sets *tokenHolder*= false, sets *receivedLI*[*next*] to false, and then sends a *Token* message to *next*. If *Q* is non-empty after sending a *Token* message to *next*, a *Request* message is sent to *next* immediately following the *Token* message so the token will eventually be returned to *i*.

- ✓ ***Procedure RaiseHeight:*** Called at non-token holding node *i* when *i* loses its last outgoing link. Node *i* raise its height using the *partial reversal* method, and inform all its neighbors of its height change with *LinkInfo* messages. All nodes on *Q* to which links are now outgoing are deleted from *Q*. If *Q* is not empty at this point, *ForwardRequest*( ) is called since *i* must send another *Request* for the token.

5. Correctness of Reverse Link Algorithm

The following theorem holds because there is only one token in the system at any time.

Theorem 1:

The algorithm ensures mutual exclusion. To prove no starvation, we first show that, after link changes cease, eventually the system reaches a "good" configuration, and then we apply a variant function argument. We will show that after link changes cease, the logical directions on the links imparted by height values will eventually form a "token oriented" DAG. Since the height values of the nodes are totally ordered, there cannot be any cycles in the logical graph, and thus it is a DAG. The hard part is showing that this DAG is token oriented, defined next.

Definition 1:

A node *i* is the *token holder* in a configuration if *tokenHolderi* = true or if a *Token* message is in transit from node *i* to *nexti* .

Definition 2:

The DAG is *token oriented* in a configuration if for every node *i, i* {0, . . . ,*n* − 1}, there exists a directed path originating at node *i* and terminating at the token holder. To prove lemma 3, that the DAG is eventually token oriented, we first show, in lemma 1, that this condition is equivalent to the absence of "sink" nodes [13], as defined below. We then show, in lemma 2, that eventually there are no more calls to *RaiseHeight*( ). Throughout, we assume that eventually link changes cease.

Definition 3:

A node *i* is a *sink* in a configuration if (*tokenHolderi* = false) and ((*myHeighti* < *heighti*[*j*]), for all *j Ni*).

Lemma 1: In every configuration of every execution, the DAG is token oriented if and only if there are no sinks.

Proof: The only-if direction follows from the definition of a token oriented DAG. If direction is proved by contradiction, Assume, in contradiction, that there exists a node i in a configuration such that *tokenHolderi* = false and for which there is no directed path starting at *i* and ending at the token holder. Since there are no sinks, *i* must have at least one outgoing link that is incoming at some other node. Since the number of nodes is finite, the network is connected, and all links are logically directed such that no logical path can form a cycle, there must exist a directed path from *i* to the token holder, a contradiction. To show that eventually there are no sinks (lemma 3), we show that there are only a finite number of calls to *RaiseHeight*( ).

Lemma 2:

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

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

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

Proof: Suppose the claim is false, and some node that holds the token after the last link change calls *RaiseHeight*( )

subsequently. Let *i* be the first node to do so. By the code, node *i* does not hold the token when it calls *RaiseHeight*(). Suppose that node *i* sends the token to neighboring node *j* at time *t*1, setting its view of *j* to be outgoing, and at a later time, *t*3, node *i* calls *RaiseHeight*( ). The reason *i* calls *RaiseHeight*( ) at time *t*3 is that it lost its last outgoing link. Thus, at time *t*2 between time *t*1 and *t*3, the link between *i* and *j* has reversed direction in *i*'s view from outgoing to incoming. By the code, the direction change at node *i* must be due to the receipt of a *LinkInfo* or *Request* message from node *j* . We discuss these cases separately below.

Case 1: *The direction change at node i is due to the receipt of a LinkInfo message from node j at time t*2. By the code, when *i* sends the token to *j* at *t*1, it sets *receivedLI*[*j*] to false. Therefore, when the *LinkInfo* message is received at *i* from *j* at time *t*2, node *i* must have already reset *receivedLI*[*j*] to true or *i* would still see the link to *j* as outgoing and would not call *RaiseHeight*( ) at time *t*2. Since *i* called *RaiseHeight*( ) after receiving the *LinkInfo* message from *j* at time *t*2, *i* must have received the *LinkInfo* message node *j* sent when it received the token from *i before* time *t*2, by the FIFO assumption on message delivery. Then node *j* must have received the token and sent it to another node, $k \_= i$, after which *j* raised its height and sent the *LinkInfo* message that node *i* received at time *t*2. However, this violates our assumption that *i* is the first node to call *RaiseHeight*( ) after the last link change, a contradiction.

Case 2: *The direction change at node i is due to the receipt of a Request message from node j at time t*2. By a similar argument to case 1, any *Request* received from node *j* would be ignored at node *i* as long as *receivedLI*[*j*] is false. But this means that node *j* must have called *RaiseHeight*() after it received the token from node *i* and subsequently sent the *Request* received by *i* at time *t*2. Again, this violates the assumption that *i* is the first node to call *RaiseHeight*( ) after the last link change, a contradiction. Therefore, node *i* will not call *RaiseHeight*( ) at time *t*2 and the claim is true. Therefore, by claim 1, there is only a finite number of calls to *RaiseHeight*( ) in any execution with a finite number of link changes. Lemma 3 follows from lemma 2, since if a node becomes a sink, it will eventually be informed via *LinkInfo* messages and will then call *RaiseHeight*( ).

Lemma 3:

Once link changes cease, the logical direction on links imparted by height values will eventually always form a token oriented DAG. Consider a node that is WAITING in an execution at some point after link changes and calls to *RaiseHeight*( ) have ceased. We first define the "request chain" of a node to be the path along which its request has propagated. Then we modify the variant function argument to show that the node eventually gets to enter the CS.

Definition 4:

Given a configuration, a *request chain* for any node *l* with a non-empty request queue is the maximal length list of node identifiers $p1 = l, p2, \ldots, pj$, where for each *i*, $1 < i \leq j$,

- *pi* 's queue is not empty,
- $pi = nextpi-1$,
- the link between $pi-1$ and *pi* is outgoing at $pi-1$ and incoming at *pi* ,
- no *Request* message is in transit from $pi-1$ to *pi*, and
- no *Token* message is in transit from *pi* to $pi-1$.

Lemma 4 gives useful information about what is going on at the end of a request chain:

Lemma 4:

The following is true in every configuration. Let *l* be a node with a non-empty request queue and let $p1 = l, p2, \ldots, pj$ be *l*'s request chain. Then

(a) *l* is in *Ql* iff *l* is WAITING,
(b) $pi-1$ is in *Qpi* , $1 < i \_ j$, and
(c) either

- *pj* is the token holder,
- or a *Token* message is in transit to *pj* ,
- or a *Request* message is in transit from *pj* to *nextpj* ,
- or a *LinkInfo* message is in transit from *nextpj* to *pj* with *nextpj* higher than *pj* ,
- or *nextpj* sees the link to *pj* as failed.

Proof: By induction on the execution. Property (a) can easily be shown to hold, since a node enqueues its own identifier when its application requests access to the CS, at which point it changes its status to WAITING. By the code, at no point will a node dequeue its own identifier until just before it enters the CS and sets its status to CRITICAL. Properties (b) and (c) are vacuously true in the initial configuration, since no node has a non-empty queue. Suppose (b) and (c) are true in the $(t-1)$st configuration, $Ct-1$, of the execution. It is possible to show these properties are true in the *t* th configuration, *Ct* , by considering in turn every possibility for the *t*th event. Most of the events applied to $Ct-1$ are easily shown to yield a configuration *Ct* in which properties (b) and (c) are true. Here we discuss the events for which the outcome is less clear by presenting the problematic cases that can appear to disrupt a request chain. We note that, in the following cases, non-token holding nodes are often required to find an outgoing link due to link reversals or failures. It is not hard to show that a node *I* that is not the token holder can always find an outgoing link due to the performance of *RaiseHeight*( ).

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

Case 1.1: *receivedLI*[*j*] is false at *i*. It must be that *i* sent the token to *j* in some previous configuration and *i* has not yet received the *LinkInfo* message that *j* must send to *i* upon receipt of the token. If the token is not in transit from *i* to *j* or held by *j* in *Ct*−1, then earlier *j* had the token and passed it on. The *Request* received by *i* was sent before the *LinkInfo* message that *j* must send to *i* upon receipt of the token. So if *j* is WAITING in *Ct*−1, it has already sent a newer *Request* and properties (b) and (c) hold for this request chain in *Ct* by the inductive hypothesis.

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

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

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

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

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

Case 1: The token holder is not in *l*'s request chain. By lemma

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

Case 3.2: Node *i* receives an indication that the link to *nexti* has failed. In *Ct* , *i* will take the same actions as it did in case 3.1, when its link to *nexti* reversed. Therefore, no action taken by node *i* can make properties (b) and (c) false and the lemma holds.

Lemma 5:

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

Lemma 6:

Vl is a variant function.

Proof: **The key points to prove are:**

(1) *Vl* never has more than *n* entries and every entry is between 1 and *n* + 1, so the range of *Vl* is well-founded.

(2) Most events can be easily seen not to increase *Vl*. Here we discuss the remaining events.

When the *Request* message at the end of *l*'s request chain is received by node *j* from node *pm*, *l*'s request chain increases in length to *m* + 1, *Vl* decreases from (*v*1, . . . ,*vm*, *n* + 1) to (*v*1, . . . ,*vm*, *v'm*+1, . .), where *v'm*+1 < *n* + 1 since *v'm*+1 is *pm*'s position in *Qj* after the *Request* message is received. When a *Token* message is received by the node *pm* at the end of *l*'s request chain, it is either

• kept at *pm*, so *Vl* decreases from (*v*1, . . . ,*vm*−1, *vm)* to (*v*1, . . ,*vm*−1, *vm* − 1), r sent toward *l*, so *Vl* decreases from (*v*1, . . . , *vm*−1, *vm)* to (*v*1, . . . ,*vm*−1), or sent away from *l*, followed by a *Request* message, so *Vl* decreases from (*v*1, . . . ,*vm*−1, *vm)* to (*v*1, . . . ,*vm*−1, *vm* − 1, *n* + 1).

(3) To see that the events that cause *Vl* to decrease will continue to occur, consider the following two cases:

5, eventually the token holder will be in *l*'s request chain.

Case 2: The token holder is in *l*'s request chain. Since no node

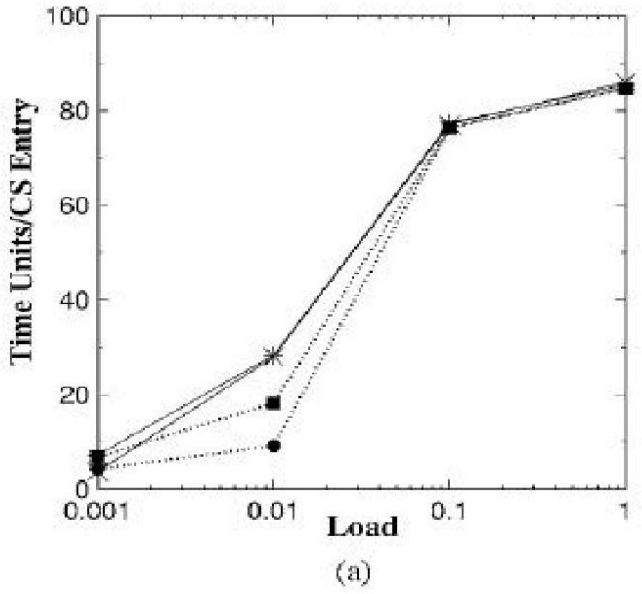
stays in the CS forever, at some later time the token will be sent and received, decreasing the value of *Vl*, by part (2) of this proof. Once *Vl* equals _1_, *l* enters the CS. We have:

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

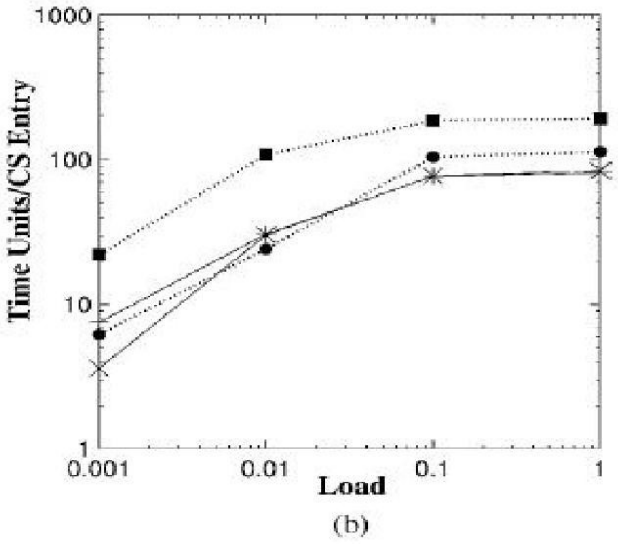
6. Simulation Results

In this section we discuss the static and dynamic performance of the Reverse Link (RL) algorithm compared to a mutual exclusion algorithm designed to operate on a static network. We simulated Raymond's token based mutual exclusion algorithm as if it were running on top of a "routing" layer that always provided shortest path routes between nodes. In this section, we will refer to this simulation as "Raymond's with routing" (RR). Raymond's algorithm was used because it is the static algorithm from which the RL algorithm was adapted and because it does not provide for link failures and recovery and must rely on the routing layer to maintain logical paths if run in a dynamic network. Complexity comparison of a routing protocol is complicated by the fact that the number of messages and amount of time needed to maintain routes can be amortized over the number of applications using those routes. In order to make our results more generally applicable, we made best-case assumptions about the underlying routing protocol used with Raymond's algorithm: that it always provides shortest paths and its time and message complexity is zero. If our simulation shows that the RL algorithm is better than the RR combination in some scenario, then the RL algorithm will also be better than Raymond's algorithm in that scenario when *any* real ad hoc routing algorithm is used. If our simulation shows that the RL algorithm is worse than the RR combination in some scenario, then it might or might not be worse in an actual situation, depending on how much worse it is in the simulation and what are the costs of the routing algorithm. A 30 node system was simulated under various scenarios. A 30 node system was chosen, in part, because for networks larger than 30 nodes the time needed for simulation was very high. Also, ad hoc networks are generally envisioned to be much smaller scale than wired networks like the Internet. Typical numbers of nodes used for simulations of ad hoc networks range from 10 to 50.

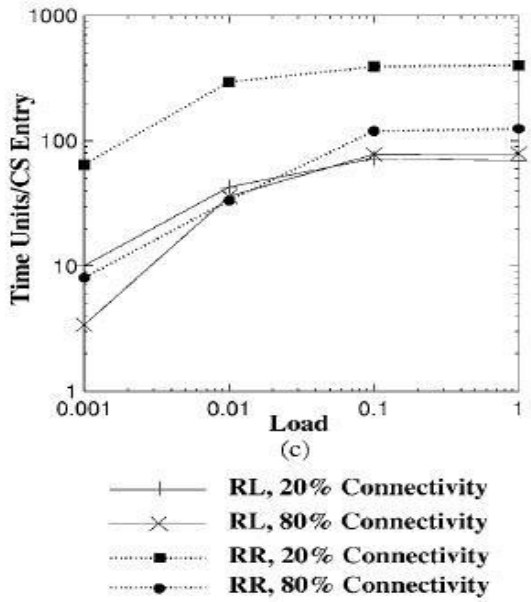
In all our experiments, each CS execution took one time unit and each message delay was one time unit. Requests for the CS were modeled as a Poisson process with arrival rate λreq. Thus the time delay between when a node left the CS and made its next request to enter the CS is an exponential random variable with mean $1/\lambda$req time units. Link changes were modeled as a Poisson process with arrival rate λmob. Hence, the time delay between each change to the graph is an exponential random variable with mean $1/\lambda$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.



(a)



(b)



7. Conclusion and Discussion

We presented a distributed mutual exclusion algorithm for mobile adhoc networks, to adapt the node mobility, and the results showing the performance of this algorithm to that of a static token based mutual exclusion algorithm running on top of an ideal ad hoc routing protocol. Here, the assumptions are no partitions in the network throughout this paper; if partitions occur, can be handled in adhoc routing protocol.

Our algorithm which uses the adhoc routing protocol generally provides better average waiting time per CS entry. Our results shows that the message complexity per CS would not be greater than the message complexity for nodes in static mobility when running in the top of the adhoc routing algorithm.

References

1. topology, IEEE Transactions on Communications C-29(1) (1981) 11–18.
2. D.M. Dhamdhere and S.S. Kulkarni, A token based *k*-resilient mutual exclusion algorithm for distributed systems, Information Processing Letters 50 (1994) 151–157.
3. R.Karthikeyan,"A Survey on Sensor Networks" in the International Journal for Research & Development in Technology Volume 7, Issue 1, Jan 2017, Page No:71-77.
4. R.Karthikeyan, & et all "Web Based Honeypots Network",in the International journal for Research & Development in Technology.Volume 7.Issue 2 ,Jan 2017,Page No.:67-73 ISSN:2349-3585.
5. R.Karthikeyan, & et all,"A Simple Transmit Diversity Technique for Wireless Communication",in the International journal for Engineering and Techniques. Volume 3. Issue 1, Feb 2017, Page No.:56-61 ISSN:2395-1303.
6. R.Karthikeyan, & et all "Strategy of Trible – E on Solving Trojan Defense in Cyber Crime Cases", International journal for Research & Development in Technology.Volume 7.Issue 1 ,Jan 2017,Page No.:167-171.
7. D.B. Johnson and D.A.Maltz, Dynamic source routing in ad hoc wireless networks, in: *Mobile Computing*, eds. T. Imielinski and H. Korth (Kluwer Academic, 1996) pp. 153–181.
8. Karthikeyan, & et all"Advanced Honey Pot Architecture for Network Threats Quantification" in the international journal of Engineering and Techniques, Volume 3 Issue 2, March 2017, ISSN:2395-1303, PP No.:92-96.
9. R.Karthikeyan, & et all "Estimating Driving Behavior by a smart phone" in the international journal of Engineering and Techniques, Volume 3 Issue 2, March 2017, ISSN:2395-1303,PP No.:84-91.
10. R.Karthikeyan, & et all "SAMI: Service- Based Arbitrated Multi-Tier Infrastructure for Cloud

Computing" in the international journal for Research & Development in Technology, Volume 7 Issue 2, Jan 2017,ISSN(0):2349-3585, Pg.no:98-102

11. R.Karthikeyan, & et all "FLIP-OFDM for Optical Wireless Communications" in the international journal of Engineering and Techniques, Volume 3 Issue 1, Jan - Feb 2017, ISSN:2395-1303,PP No.:115-120.

12. R.Karthikeyan, & et all "Application Optimization in Mobile Cloud Computing" in the international journal of Engineering and Techniques, Volume 3 Issue 1, Jan - Feb 2017, ISSN:2395-1303,PP No.:121-125.

13. R.Karthikeyan, & et all "The Sybil Attack" in the international journal of Engineering and Techniques, Volume 3 Issue 3, May - Jun 2017, ISSN:2395-1303,PP No.:121-125.

14. R.Karthikeyan, & et all "Securing WMN Using Hybrid Honeypot System" in the international journal of Engineering and Techniques, Volume 3 Issue 3, May - Jun 2017, ISSN:2395-1303,PP No.:121-125.

15. R.Karthikeyan, & et all "Automated Predictive big data analytics using Ontology based Semantics" in the international journal of Engineering and Techniques, Volume 3 Issue 3, May – Jun 2017, ISSN:2395-1303,PP No.:77-81.

16. R.Karthikeyan, & et all "A Survey of logical Models for OLAP databases" in the international journal of Engineering and Techniques, Volume 3 Issue 3, May - Jun 2017, ISSN:2395-1303,PP No.:171-181.

17. R.Karthikeyan, & et all "A Client Solution for Mitigating Cross Site Scripting Attacks" in the international journal of Engineering Science & Computing, Volume7,Issue6, June 2017, ISSN(0):2361-3361,PP No.:13063-13067.

18. I. Keidar and D. Dolev, Efficient message ordering in dynamic networks, in: *Proc. of 15th Annual Symp. on Prin. of Dist. Computing* (1996) pp. 68–76.

19. 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* (1998) pp. 66–75.

20. R.Karthikeyan, & et all "A Condensation Based Approach to Privacy Preserving Data Mining" in the international journal of Engineering Science & Computing, Volume7,Issue6, June 2017, ISSN(0):2361-3361,PP No.:13185-13189.

21. R.Karthikeyan, & et all "Biometric for Mobile Security" in the international journal of Engineering Science & Computing, Volume7,Issue6, June 2017, ISSN(0):2361-3361,PP No.:13552-13555.

22. R.Karthikeyan, & et all "Data Mining on Parallel Database Systems" in the international journal of Engineering Science & Computing, Volume7,Issue7, July 2017, ISSN(0):2361-3361,PP No.:13922-13927.

23. R.Karthikeyan, & et all "Ant Colony System for Graph Coloring Problem" in the international journal of Engineering Science & Computing, Volume7,Issue7, July 2017, ISSN(0):2361-3361,PP No.:14120-14125.

24. R.Karthikeyan, & et all "Classification of Peer –To-Peer Architectures and Applications" in the international journal of Engineering Science & Computing, Volume7,Issue8, Aug 2017, ISSN(0):2361-3361,PP No.:14394-14397.

25. R.Karthikeyan, & et all "Mobile Banking Services" in the international journal of Engineering Science & Computing, Volume7,Issue7, July 2017, ISSN(0):2361-3361,PP No.:14357-14361.

26. P. Krishna, N.H. Vaidya, M. Chatterjee and D.K. Pradhan, A clusterbased approach for routing in dynamic networks, in: *Proc. of ACM SIGCOMM Computer Communication* .

27. R.Karthikeyan, & et all "Neural Networks for Shortest Path Computation and Routing in Computer Networks" in the international journal of Engineering and Techniques, Volume 3 Issue 4, Aug 2017, ISSN:2395-1303,PP No.:86-91.

28. R.Karthikeyan, & et all "An Sight into Virtual Techniques Private Networks & IP Tunneling" in the international journal of Engineering and Techniques, Volume 3 Issue 4, Aug 2017, ISSN:2395-1303,PP No.:129-133.

29. R.Karthikeyan, & et all "Routing Approaches in Mobile Ad-hoc Networks" in the International Journal of Research in Engineering Technology, Volume 2 Issue 5, Aug 2017, ISSN:2455-1341, Pg No.:1-7.

30. M.L. Neilsen and M. Mizuno, A DAG-based algorithm for distributed mutual exclusion, in: *Proc. of Intl. Conf. on Dist. Comp. Systems* (1991) pp. 354–360.