A Leader based $k$-Local Mutual Exclusion Algorithm using Token for MANETs

ASHISH KHANNA, AWADHESH KUMAR SINGH AND ABHISHEK SWAROOP
MAIT, Delhi-110085, India
NIT, Kurukshetra-136119, Haryana, India
Sharda University, Greater Noida, U.P., India
E-mail: ashishk746@yahoo.com, aksinreck@rediffmail.com, asa1971@gmail.com

The $k$-local mutual exclusion is a generalization of local mutual exclusion problem introduced by Attiya et al. In $k$-local mutual exclusion, it is assumed that the $k$ identical copies of a resource are shared among the geographically close nodes. The paper proposes a solution to the $k$-local mutual exclusion problem in MANETs. The algorithm uses a leader-based approach and the leader is equipped with a token. It is suited to handle mobility that triggers the dynamism in topology of ad hoc networks. The algorithm satisfies safety, starvation freedom and $l$-deadlock avoidance properties. The best case message complexity of our algorithm is $O(1)$ whereas the worst case message complexity is $O(N)$. To the best of our knowledge, it is the first algorithm to solve $k$-local mutual exclusion problem in MANETs. The solution to token loss problem is also included in the present exposition.

Keywords: Ad hoc Network, Local Mutual Exclusion, Neighborhood, Resource allocation

1. INTRODUCTION

In distributed computing environment, there are many resources that are shared among multiple processors. Thus, a number of processors often compete to have exclusive access to such shared resources. In order to avoid conflict, we need sophisticated algorithm to control the access to such resource. Such algorithms are called mutual exclusion algorithms. Hence, mutual exclusion is a classic computing problem [1]. A process is called waiting or hungry when it is trying to get the resource. Once it gets access to the resource it is called eating or executing in critical section (CS). Afterwards, the process relinquishes the control over the shared resource, called exit from critical section. The mutual exclusion algorithms are broadly of two types [2], namely token based where token holder has privilege to enter CS and non-token based, a.k.a. permission based, where the process permitted by all other processes is allowed to enter CS. For static networks, many variants of mutual exclusion problem have been proposed in the literature, e.g., $k$-mutual exclusion [3] and group mutual exclusion [4]. Algorithm presented in this paper is a token based.

The mobile ad hoc network (MANET) is inherently resource constrained environment due to dynamic topology, limited bandwidth, limited battery power and low processing capability etc. Therefore, the algorithms designed for mutual exclusion in static networks cannot be directly applied in MANETs. Hence, many algorithms have been proposed in literature to handle mutual exclusion in MANETs [5, 6, 7, 8]. Sharma-
Bhatia-Singh [9] presented a detailed survey of the mutual exclusion in MANETs.

The mutual exclusion problem is to ensure that only one of the contending nodes is allowed to enter CS. The $k$-mutual exclusion problem [3] is a generalization of the mutual exclusion problem in which there are either $k$ identical copies of a shared resource or the resource itself is of type that allows access to $k$ processes simultaneously. Therefore, at most $k$ nodes may enter CS simultaneously.

The dining philosopher’s problem [10, 11] is an interesting variant of mutual exclusion problem in static networks. Each node cycles through thinking, hungry, eating, and exit states; thinking state signifies that the node is not interested in the resource. Although, the solution to dining philosopher’s problem works as a solution to mutual exclusion problem, it has a special property that the failure of a node does not affect the entire system. In other words, the effect of node failures is limited. In ad hoc networks, most of the activities are local, thus, effect of any failure is also local. This characteristic of ad hoc network is analogous to the above mentioned property of dining philosophers set up. Thus, dining philosopher’s problem has been solved in ad hoc networks under the title local mutual exclusion problem by Attiya et al. [12]. Therefore, unlike conventional mutual exclusion, local mutual exclusion algorithms have better failure locality [13]. It is one of the highly desired features in the resource allocation algorithms. In local mutual exclusion, no two neighboring nodes can enter CS simultaneously; however, the nodes which are not neighbors can be in their CS simultaneously. On the other hand, in classical or global mutual exclusion no two nodes (how far apart) can be in CS simultaneously.

Attiya-Kogan-Welch [13] has indicated that in MANETs, the local mutual exclusion problem has more potential applications in comparison to global mutual exclusion problem. The example applications of local mutual exclusion problem in MANETs are sharing a specialized hardware in a region, implementation of virtual mobile node (VMN) [14] etc.

Another extension of the local mutual exclusion problem in MANETs could be a situation in which there could be more than one, say $k$, instances of identical resource which can be shared by the neighboring nodes; therefore, at most $k$ nodes can access the $k$ instances of the shared resource simultaneously. We name this problem as $k$-local mutual exclusion problem (called KLME, henceforth). The KLME problem is different from $k$-mutual exclusion problem in the sense that in $k$-mutual exclusion problem there are $k$ instances of the resource in the whole system where as in $k$-local mutual exclusion problem there are $k$ instances of the shared resource in each neighborhood.

When a mobile node in CS changes its neighborhood, it leaves the resource instance allocated in the previous neighborhood and a new resource instance is allocated in the new neighborhood in order to avoid the application discontinuity. Alternatively, the mobile switches from ‘eating’ state to ‘hungry’ state bypassing the ‘exit’ in between, e.g. in case of seamless handover, the ongoing call is allocated new channel when the node moves into new neighborhood and new links are created [13]. Such type of application is possible using local mutual exclusion and its generalization KLME, proposed here. However, in case of multiple K-mutual exclusion (KME) switching from ‘eating’ state to ‘hungry’ state is prohibited for any node without passing through ‘exit’ state.

The neighborhood can be defined as the independent smaller unit of a larger geographical area. The larger area is divided in smaller neighborhoods based upon the
location of shared resources. Each set of shared resources placed at one location can be accessed in a particular neighborhood only. For example, the shared resource may be a shared database, dedicated wireless channel, satellite uplink facility and wireless printer. Therefore, the area in which the resource can be used is limited and can be considered as its neighborhood. One interesting example of neighborhood concept is a building or university blocks where each floor or a block can have resource centre of \( k \) identical copies (\( k \) identical replicas of Database, \( k \) dedicated wireless channels) and each mobile node like PDA’s (personal digital assistant) or a laptop or any other mobile device can move from one floor to another, therefore each floor or block can be considered as neighborhood. The complete building can also have a one common resource centre dedicated to whole building where we can apply any \( k \) mutual exclusion algorithm and to deal with these special neighborhoods we need a specialized algorithm or concept where these small neighborhoods can be handled in different manner than the conventional \( k \)-mutual exclusion approach. Conventional \( k \)-mutual exclusion is different from the \( k \)-local mutual exclusion algorithm in the same way as local mutual exclusion algorithm given by Attaiya et al. [13] is different from conventional mutual exclusion algorithm. Moreover, one neighborhood may contain \( k \) instances of a resource. In this situation, \( k \)-local mutual exclusion algorithm is required to ensure the efficient and correct sharing of the resources. In the following Fig. 1, in the system, there are several mobile nodes divided in to four neighborhoods and each neighborhood contains multiple instances of resource type R1. A copy of \( k \)-local mutual exclusion algorithm is running in each neighborhood. The mobile nodes may move from one neighborhood to another and an outgoing node using one instance of resource R1 in current neighborhood (state is eating), may enter in hungry state in the new neighborhood without moving to thinking state.

Any solution to the \( k \)-local mutual exclusion problem must also satisfy the requirements for \( k \)-mutual exclusion [15].

1. **Safety:** At most \( k \) nodes of the neighborhood may be in CS simultaneously.
2. **\( l \)-deadlock avoidance:** If the number of nodes in CS in a neighborhood is less than \( k \) and some nodes in the neighborhood are requesting to enter CS then at least one such node will be allowed to enter CS with in a finite time without waiting for any other node to leave CS.
3. **Starvation freedom:** Each requesting node will eventually enter CS.

The rest of the paper has been organized in the following manner. Section 2 contains the related work and section 3 contains the system model and working of algorithm. The correctness proof and the performance analysis of the algorithm have been discussed in section 4 and section 5 respectively. Section 6 includes simulation results under various required conditions. The token loss problem has been handled in section 7. Finally, Section 8 concludes the work and contains future research directions.
Fig. 1. Multiple Neighborhoods

2. RELATED WORK

The mutual exclusion problem is an important problem in any type of distributed system. The first distributed algorithm for solving the mutual exclusion problem in MANETs was proposed by Walter-Kini [5] which uses a directed acyclic graph (DAG) of token-oriented pointers, maintaining multiple paths leading to the node holding the token. The algorithm is well suited to the distributed mobile setting because it requires nodes to keep information only about their immediate neighbors. Walter-Welch-Vaidya [6] proposed an alternative algorithm (named as RL). In their algorithm, the requests are forwarded to the token holder over the tree. The token is delivered over the reverse tree path to the requesting node. The token holder will always be the lowest node in the DAG. The algorithm proposed by Roberto-Baldoni-Virgillito [7] is based on a dynamic logical ring and combines the two methods token-asking and circulating token. The algorithm aims to maintain device power consumption as low as possible by reducing the number of hops traversed per CS execution and by avoiding sending any control message when no process requests the CS. Derahab-Badache [8] proposed a new architecture to solve the problem of mutual exclusion for MANETs. The architecture is structured into two layers (i) a service layer that contains a distributed mutual exclusion algorithm and (ii) a network layer that includes two routing protocols.

The algorithms for variants of classical mutual exclusion problem such as $k$-mutual exclusion [16, 17, 18], group mutual exclusion [19, 20] and $h$-out of $k$-mutual exclusion [21] for MANETs have also been proposed in the literature.

Walter-Cao-Mohanty [16] presented fault-tolerant token based distributed $k$-mutual exclusion algorithm which adjusts to node mobility. The algorithm has been specifically designed to suit ad-hoc environment, since; it requires nodes to communicate with only
their current neighbors. 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. Masum et. al. [18] presented a consensus-based mobility aware l-Exclusion (LE) algorithm that operates asynchronously and copes explicitly with arbitrary (possibly concurrent) topology changes associated with such networks. The algorithm is consensus based and a mobile node intending to enter CS tries to collect enough consensuses, and uses diffusing computations for this purpose. This paper presents a simulation to demonstrate that the proposed algorithm, as compared to the $k$-Reverse Link (KRL) algorithm, is quite effective to variety of operating conditions.

A different version of classical mutual exclusion problem is known as local mutual exclusion in which the nearby nodes (i.e. nodes in the same neighborhood) compete with each other for the exclusive access of a shared resource. The local mutual exclusion problem has been discussed earlier in static networks [22, 23], however, Attaiyya et al. [12] solved local mutual exclusion problem first time in MANETs. They claimed that the local mutual exclusion problem has more potential applications than the classical mutual exclusion problem especially in MANETs. They proposed two algorithms using doorways for solving local mutual exclusion problem in MANETs.

In recent years, several researchers focused on solving mutual exclusion problem and its variants [24, 25, 26, 27, 28, 29, 30] in MANETs, however, the problem of local mutual exclusion and its variants are still unexplored despite having several possible applications. Hence, in the present paper, the concept of local mutual exclusion is further extended to $k$-local mutual exclusion in MANETs. Moreover, a leader based solution using token to solve $k$-local mutual exclusion problem in MANETs has also been proposed. The frequent change of leadership in the proposed algorithm mitigates the problem of single point failure to a large extent.

3. SYSTEM MODEL

We consider a MANET, in which there are $n$ mobile nodes and the resources to be shared are located at different places and each set of resource (containing $k$ identical resources) has a transmission range over which it can be accessed by mobile nodes. The area in which a particular set of resources can be accessed defines a neighborhood. The mobile nodes in a neighborhood compete for the resources available in the neighborhood.

1. There is no assumption on the processing speed of a mobile node. Moreover, the message propagation delay is finite but unpredictable.
2. A mobile node cannot make a new request until its old request is satisfied.
3. A workable link layer protocol is in place.
4. No node will leave its neighborhood area while in CS. If a node executing in CS wants to exit the neighborhood, it will come out of CS before leaving the neighborhood.
5. Underlying ad hoc network is of QUASI stable nature.
6. Message delivery is ensured between two connected nodes via a routing protocol.
7. Each node has unique identifier.
8. Number of nodes in the system is finite.
The same assumptions have been made by previous works on MANETs [6, 13].

3.1 The Algorithm Concept

In this section, the brief working of the proposed algorithm for \(k\)-local mutual exclusion problem in MANETs has been discussed. To the best of our knowledge, it is the first algorithm to solve \(k\)-local mutual exclusion problem. The algorithm uses the concepts of token and leader.

After initializing all nodes in neighborhood area, the leader election process is initiated. The node which has been elected as leader generates a token and broadcasts its leadership information. The token contains a variable \(K_{\text{free}}\) which represents the number of free resources at a particular instance of time. When neighboring nodes receives the token holder information they can send their request to the main token holder, if they want to enter CS.

When the node which has been initially elected leader, generates a request, it enters CS without any delay. On the other hand, if the leader receives first request from some other node, it transfers the token to the requesting node and broadcasts the new leader information. Each time the token is transferred to a new leader, token leader number is incremented by one, and this information is also broadcasted along with the new leader information which helps in rejecting the stale new leader information. While the token holder node is in CS and it receives a request from some other node, it permits the requesting node to enter CS, in case a free resource is available. Otherwise, the request is added in the token queue and the requesting node is acknowledged about the receiving of the request. When a node which has been permitted by the leader to enter CS comes out of CS, it informs the leader about its exit from CS. On receiving this information, the leader increments the number of free resources and permits another node from its request queue only if such a node exists.

In case, the leader comes out of CS, it checks if the token queue is empty. If so, it holds the token and waits for a new request before transferring the token. On the other hand, if there are some pending requests in the token queue, the node at the front of the token queue is selected as new leader, the token is transferred to it and the leader information is broadcasted simultaneously.

If the request is received by a node after it has transferred the token to some other node, the node stores the request in a queue. When this node again receives the token it checks this list and adds any unfulfilled request in the token request list. On receiving the new leader information, if the node is in waiting state and the request acknowledgement has yet been not received, the request message is retransmitted.

When a leader comes out of CS the token is passed to the node at the front of the token request queue, however, if token request queue is empty then the leader holds the token. In case a non leader node comes out of CS, it sends the release message to the leader according to the latest information stored at the node. When a leader receives the release message then the corresponding entry is deleted from the token resource allocation list. Moreover, when token is transferred, the new leader information is also broadcasted by the old leader that includes a resource allocation list, its node id and token leader number. Therefore, the node receiving the information can check its entry in this list and retransfer the release to the new leader again if required. On the other hand,
if the node receiving the release message has passed the token to some other node, then that release message information along with the request number is stored in a local list. When this node again receives the token at some later stage this list is compared with the list held by the token and the number of free resources and other data structures are updated accordingly. Moreover, if the request queue is not empty then permission to enter CS is sent to the requesting node(s) in case free resource(s) is (are) available.

A node can leave the neighborhood and also can join the neighborhood. A outgoing token holder node executing in CS sends the token either to the first requesting node or to the lowest id node present in the neighborhood executing in its remainder section in case the requesting queue is empty. Moreover, the outgoing node broadcasts the leaving message along with pending request and any pending release information. Additionally, if a node is in CS state just before moving out of the neighborhood, it enters new neighborhood in waiting state.

3.2 Data Structure

3.2.1 Data Structure at each node i

M_T: Boolean variable indicating whether the node has token or not.
RN: Request number of node i.
State: Indicates the state of node (CS-Critical section, W-Waiting, R-Remainder).
Node_req_list: Stores the request number and node ids of neighbouring nodes.
M_Th_id: Id of node which is currently holding the token according to node i.
Req_ack: Boolean variable indicating whether request acknowledgement is received.
Node_release_list: Contains node id and current request number from which the release message has been received.
Node_leader_no: Contains the value that how many times new leader has been elected according to node i.

3.2.2 Token: The token contains the following data structures.

T_alloc_id: An array of length k, each entry contains id of the nodes to which the permission to enter CS has been granted with their request number.
T_req_list: Stores the requests number of all nodes in neighbourhood.
K_free: Indicates the number of free resources available at any point of time.
R_Queue: Stores the requests which have reached the token holder node but not satisfied.
Token_leader_no: Count of the leader changed events.

3.3 Messages

1. Leader_info(i,T_alloc_id,Token_leader_no): node i broadcasts leader id along with other information to all neighboring nodes.
2. I_am_here(i): On joining a neighborhood, node i broadcasts this message to all neighborhood nodes.
3. Req_CS(i,RN): Sent by the node i to node which is the current leader according to i.
4. Token (i, T_alloc_id, T_req_list, K_free, R_Queue): A node receiving the Token becomes the leader.
5. **Req_ack()**: Token holder node sends this message to requesting node on receiving its request.
6. **Per_grant()**: sent as a permission to enter CS by token holder to requesting node.
7. **Release (i, RN_i)**: sent to the leader after a non-leader node comes out of CS.
8. **I_am_leaving(i,node_release_list_i, Node_release_list)**: broadcasted by a node just before leaving the neighborhood.

### 3.4 Event Driven Description of the Algorithm executed at node i.

**Initialization**

```plaintext
Initialize
for(i=1 to n) // where n is number of nodes in neighborhood
    M_T_i=0; State_i=R; M_Th_id = Ø; Req_ack_i = 0
for (j=1 to n)
    Node_req_list_i[j] = 0; Node_release_list_i[j]=-1
    Node_leader_no_i = 0
call Generate Token()
```

**GenerateToken()**

Assume a arbitrary node i in neighbourhood as leader.

```plaintext
for(i=1 to K) //K is total no. of resources
    T_alloc_id[i]= {-1, 0}
    Token_leader_no = 1
    M_T_i = 1; M_Th_id_i = i;Broadcast Leader_info to all nodes;
    T_req_list is initialized with 0 for all nodes
    K_free = K; R_Queue = Ø;
```

**A. On receiving Leader_info(j, T_alloc_id,Token_leader_no)**

```plaintext
    M_Th_id_i = j;
    if (Token_leader_no > node_leader_no_i)
        node_leader_no_i = Token_leader_no
        if (State_i = R && i ∈ T_alloc_id)
            send release to j
        if (State_i = W && Req_ack_i ≠ 1 )
            send Req_CS to M_Th_id
    else
        reject the message
```

If the received leader information is stale, it is rejected. Otherwise, the leader information at node i is updated. If node i is in remainder section and according to leader one instance of resource is allocated to node i, the release message has to be sent again. Additionally, if node i is in waiting state and its request has not been acknowledged yet, the request is resent to the leader.

**B. Node i is requesting**

```plaintext
    State_i = W; Req_ack_i = 0; RN_i=RN_i+1
    Node_req_list_i[i] = Node_req_list_i[i]+1
```
if (M_Th_id[i] = i)
    Req_ack[i] = 1; K_free = K_free - 1; Enter CS; Exit CS
elseif (M_Th_id[i] = Ø)
    Node i will wait for Leader_info message
else
    Send Req_CS to M_Th_id[i]

When node i moves to requesting state there are three possibilities. (a) Node i is leader. The token holder will enter CS immediately. (b) Node i is not leader and does not have information of leader. Node i will wait for leader information. (c) Node i is not leader, however, possess leader information. It will send the request message to the leader.

C. On receiving Req_CS (j, RN_j) from node j
   if (Node_req_list[i][j] < RN_j)
       if (M_Th_id[i] ≠ i)
           Node_req_list[i][j] = RN_j //Storing request in node request list if Token has left
           else
               if (j ∈ T_alloc_id)
                   remove the corresponding entry;
                   K_free = K_free + 1
                   T_req_list[j] = RN_j; Node_req_list[j][j] = RN_j
                   Send Req_ack to node j
                   if (State[i] = R) // node i is holding idle token
                       Token_leader_no = Token_leader_no + 1
                       Broadcast Leader_info message
                       Send Token to requesting node j
                       M_th_id[i] = j; M_t_i = 0
                   elseif (K_free > 0) // request queue is empty
                       K_free = K_free - 1; Send Per_grant to node j
                       (j,RN_j) is added in T_alloc_id
                   else
                       Append the request in R.Queue

   If the request is stale it is rejected. Now, if Node i is not leader, the request is added in the node request queue. In case, node i is leader the request is acknowledged. If node j is occupying one instance of resource according to node i, that resource is freed. If leader is holding idle token, the token is transferred to requesting node and new leader information is broadcasted. On the other hand, if leader is in CS with free resources available, the leader permits the node j to enter CS, otherwise the request will be appended in the token queue.

D. On receiving Req_ack() from leader
   Req_ack[i] = 1

E. On receiving per_grant() from node j
State, \( i \) = CS; Enter CS; Exit from CS

**F. Node i exit from CS**

State, \( i \) = R;
if (\( i = M_\text{Th}_{id_i} \))
\[ K_{free} = K_{free} + 1 \]
if (\( R\_Queue \neq \emptyset \))
\[ M_{T_i} = 0; \text{Token_leader_no} = \text{Token_leader_no} + 1; \]
\[ T\_alloc_id = T\_alloc_id \setminus \{i, RN_i\} \]
Send Token to the node at front of \( R\_Queue \);
Broadcast \( \text{Leader_info} \) message; \( M_{Th_id_i} = X \)
else
Send Release to \( M_{Th_id_i} \)

When a node exits from CS its state is set to remainder. If the exiting node is leader, one instance of the resource is freed. Moreover, if token queue is not empty the token is transferred to the node at the front of token queue and broadcast the leader information. On the other hand, if the exiting node is not leader, the release message is sent.

**G. On receiving receives release \((j,RN)\) from node j**

if (\( M_{th_id_i} = i \))
if (\( j \in T\_alloc_id \))
\[ T\_alloc_id = T\_alloc_id \setminus \{j, RN_j\}; K_{free} = K_{free} + 1 \]
if (\( R\_Queue \neq \emptyset \))
Send \( \text{Per_grant} \) to node X // X is at front of \( R\_Queue \)
else
reject the message // stale release
else // Token has left for new leader
add \((j,RN_j)\) in \( \text{Node_release_list}_i \)

In case, the node which receives release message is token holder and resource is allocated to the sender node, sender node is removed from the list and one instance of resource is freed. If there is any pending request(s) in token queue, the freed resource is allocated to the requesting node at front of token queue. On the other hand, if non leader node receives the release message it stores the same in its local data structure.

**H. On receiving Token \((i, T\_alloc_id, T\_req_list, K_{free}, R\_Queue)\)**

\[ M_{T_i} = 1; M_{Th_id_i} = i \]
Compare \( \text{Node_req_list}_i \) with \( \text{T_req_list} \) and if any pending request(s) is (are) found add it to \( R\_Queue \), also update \( \text{Node_req_list}_i \) and \( \text{T_req_list} \) for each \( j \in T\_alloc_id \)
if (\( j \in \text{Node_release_list}_i \))
\[ T\_alloc_id = T\_alloc_id \setminus \{j, RN_j\}; K_{free} = K_{free} + 1; \text{Remove } j \text{ from} \]
\( \text{Node_release_list}_i \)
if (\( \text{State}_i = W \))
\[ K_{free} = K_{free} - 1; \text{Enter CS; Exit CS;} \]
if (K_{free} > 0) 
    allocate the remaining resources to the nodes in token queue 
else 
    if (R_{Queue} ≠ Ø) 
        send Token to node at front of R_{Queue} 
    else 
        hold idle Token 

On receiving the token, the token queue is updated with the local request list. Moreover, any pending release received by node i before receiving the token is taken into account. Now, if node i is waiting, it enters critical section and if still free resources are available and token queue is not empty resources are allocated as per FIFO policy. Otherwise, the node either holds the idle token (token queue is empty) or transfers the token to the node at the front of the token queue.

\textbf{I. Node i is about to leave the neighborhood area}
if (i = M_{Th_id}, && State_i = CS) 
    K_{free} = K_{free} + 1; remove i from T_{alloc_id}; state_i=W 
    if (R_{Queue} ≠ Ø) 
        send Token to node at front of R_{queue} 
    else 
        send Token to that lowest id which is neither in T_{alloc_id} and also present inside neighborhood; Broadcast leader info message 
elseif (state_i=CS) 
    send release to M_{Th_id}; state_i=W 
    Broadcast I_am_leaving message 

When a leaving node is in CS and is token holder then it will send the token either to the requesting node at the front of token queue or to the lowest id node which is not in CS (in case request queue is empty). If node is in CS but not token holder, release message is sent to the main token holder. Moreover, if the node is in CS it enters the new neighborhood with state hungry so that the service may be continued without interruption if a free resource is available. The information about new leader is broadcasted. The node will delete all local information before leaving the neighborhood. A message indicating that the node is leaving the neighborhood is broadcasted.

\textbf{J. Node i receives I_am_leaving (j,node_req_list,node_release_list)}
Update the node_req_list_i and node_release_list_i 
Delete its entry from all lists at node i 
if (i = M_{Th_id}) 
    Delete j from all lists on Token and take corresponding actions. 
    if (i ∈ T_{alloc_id}) 
        remove the node i from the T_{alloc_id}.

Upon receiving this message, the receiving nodes delete all the information related to node j and updates their data structure accordingly. In case, this information is
received by the leader and if the leader has allocated any resource to this node the resource is assumed to be free.

**K. New node i entering the neighbourhood.**

send I_am_here to all nodes in neighbourhood
wait for leader_info; M_Th_id_i = -1
Node_req_list_i = 0 for all entries Req_ack_i = 0

**L. Node i receives I_am_here(j, state_j)**

add entry (j,0) to Node_req_list_i;
if (M_Th_id_i = i)
    send Leader_info to node j
    if (state_j=W) act as if i has received request from node j

### 3.5 An Illustration

In order to explain the working of the proposed algorithm we consider the following scenario.
Initially the number of nodes in neighborhood = 4 \{1, 2, 3, 4\}
Number of free resources in neighborhood \( K = 2, K_{\text{free}} = K \)
The table 1 shows the static execution of the algorithm and changes in data structure after the execution of events.

- **a.** Initialization, node 2 is elected as leader after leader election process, Leader_info is broadcasted to all nodes and Leader_info received by all nodes.
- **b.** Node 3 requests for CS and sends its request to leader, leader sends Req_ack, Req_ack received by node 3, Token transferred and Leader_info broadcasted, Token received by node 3 and node 3 enters CS.
- **c.** Node 4 wants to enter CS while node 3 is in CS. Node 4 sendsReq_CS to node 3, node 3 receives Req_CS, sends Per_grant to node 4, node 4 receives Per_grant and enters CS.
- **d.** Node 5 enters the neighborhood, node 5 sends I_am_here message which is received by all nodes, node 3 sends Leader_info and node 5 receives Leader_info.
- **e.** Node 4 exits from CS, sends Release message to leader, Release message received by leader.
- **f.** Node 4 goes out from neighborhood area after sending I_am_leaving, I_am_leaving is received by all nodes.
- **g.** Node 3 exits from CS, node 1 sends Req_CS to leader, node 3 sends token to node 1 and broadcast new Leader_info to all nodes which is received by all nodes.
- **h.** Node 5 sends Req_CS, node 2 sends Req_CS, both requests received by leader in order, node 5 receives Per_grant and node 2 request added in R_Queue.
Table 1. An Illustration

<table>
<thead>
<tr>
<th>State/Time</th>
<th>R_Queue</th>
<th>T_alloc_id</th>
<th>State(s.p)</th>
<th>M_Th_id</th>
<th>Node_Req_List(r.P)</th>
<th>Kfree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event a occurred</td>
<td>Empty</td>
<td>-1,0</td>
<td>s.P1=R</td>
<td>l.P1 = 2</td>
<td>r.P1 = 0000</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 2</td>
<td>r.P2 = 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P1=R</td>
<td>l.P1 = 2</td>
<td>r.P1 = 0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=R</td>
<td>l.P4 = 2</td>
<td>r.P4 = 0000</td>
<td></td>
</tr>
<tr>
<td>Event b occurred</td>
<td>Empty</td>
<td>3,1</td>
<td>s.P1=R</td>
<td>l.P1 = 3</td>
<td>r.P1 = 0010</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 3</td>
<td>r.P2 = 0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=CS</td>
<td>l.P3 = 3</td>
<td>r.P1 = 0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=R</td>
<td>l.P4 = 3</td>
<td>r.P2 = 0010</td>
<td></td>
</tr>
<tr>
<td>Event c occurred</td>
<td>Empty</td>
<td>3,1</td>
<td>s.P1=R</td>
<td>l.P1 = 3</td>
<td>r.P1 = 0010</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 3</td>
<td>r.P2 = 0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=CS</td>
<td>l.P3 = 3</td>
<td>r.P3 = 0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=CS</td>
<td>l.P4 = 3</td>
<td>r.P4 = 0011</td>
<td></td>
</tr>
<tr>
<td>Event d occurred</td>
<td>Empty</td>
<td>3,1</td>
<td>s.P1=R</td>
<td>l.P1 = 3</td>
<td>r.P1 = 00100</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 3</td>
<td>r.P2 = 00100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=CS</td>
<td>l.P3 = 3</td>
<td>r.P3 = 00110</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=CS</td>
<td>l.P4 = 3</td>
<td>r.P4 = 00111</td>
<td></td>
</tr>
<tr>
<td>Event e occurred</td>
<td>Empty</td>
<td>3,1</td>
<td>s.P1=R</td>
<td>l.P1 = 3</td>
<td>r.P1 = 00100</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 3</td>
<td>r.P2 = 0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=CS</td>
<td>l.P3 = 3</td>
<td>r.P3 = 00110</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=R</td>
<td>l.P4 = 3</td>
<td>r.P3 = 00110</td>
<td></td>
</tr>
<tr>
<td>Event f occurred</td>
<td>Empty</td>
<td>3,1</td>
<td>s.P1=R</td>
<td>l.P1 = 3</td>
<td>r.P1 = 0010</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 3</td>
<td>r.P2 = 0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=CS</td>
<td>l.P3 = 3</td>
<td>r.P3 = 0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=R</td>
<td>l.P4 = 3</td>
<td>r.P4 = 0010</td>
<td></td>
</tr>
<tr>
<td>Event g occurred</td>
<td>Empty</td>
<td>1,1</td>
<td>s.P1=CS</td>
<td>l.P1 = 1</td>
<td>r.P1 = 1010</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=R</td>
<td>l.P2 = 1</td>
<td>r.P2 = 1010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=R</td>
<td>l.P3 = 1</td>
<td>r.P3 = 1010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=R</td>
<td>l.P4 = 1</td>
<td>r.P4 = 1010</td>
<td></td>
</tr>
<tr>
<td>Event h occurred</td>
<td>2 is at the front of R_Queue</td>
<td>1,1</td>
<td>s.P1=CS</td>
<td>l.P1 = 1</td>
<td>r.P1 = 1111</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P2=W</td>
<td>l.P2 = 1</td>
<td>r.P2 = 1110</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P3=R</td>
<td>l.P3 = 1</td>
<td>r.P3 = 1010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s.P4=CS</td>
<td>l.P4 = 1</td>
<td>r.P4 = 1011</td>
<td></td>
</tr>
</tbody>
</table>
4. CORRECTNESS PROOF

4.1 Safety

Lemma 1: The value of $K_{\text{free}}$ never exceeds $k$.

Proof: Initially the value of $K_{\text{free}} = k$ (from initialization)

Now let us assume that at any instance of time our proposition holds. The events which are changing the value of $k$ are as follows:

a) The token holder enters CS. $K_{\text{free}} = K_{\text{free}} - 1$

b) The token sends permission to one requesting node. $K_{\text{free}} = K_{\text{free}} - 1$

c) The token holder node comes out of CS. $K_{\text{free}} = K_{\text{free}} + 1$

d) The token holder receives valid release message. $K_{\text{free}} = K_{\text{free}} + 1$

e) Node $i$ receives token and has earlier received a valid release message from the node whose corresponding entry is in array $T_{\text{alloc_id}}$. $K_{\text{free}} = K_{\text{free}} + 1$

Now the value of $K_{\text{free}}$ is incremented by one, only in events c, d and e. However, event c must be preceded by event a, event d and event e must be preceded by event b moreover, in both events b and a, the value of $K_{\text{free}}$ is decremented by one hence nullifying the effect of event c, d and e correspondingly. Therefore, it is proved that the value of $K_{\text{free}}$ will never exceed $k$.

Lemma 2: The value of $K_{\text{free}}$ never becomes negative.

Proof: The leader only permits a node to enter CS only when value of $K_{\text{free}} >= 1$ otherwise request is added in token queue. Moreover, after permitting, $K_{\text{free}}$ is decremented only by 1. Therefore, the value of $K_{\text{free}}$ can never be less than zero.

Theorem 1: At any point of time at most $k$ nodes may be in CS.

Proof: Let us assume the contrary that there are $x$ ($>k$) nodes which are in CS simultaneously. Now the value of $K_{\text{free}} = y$ ($y >= 0$ from lemma 2)

Suppose all $x$ nodes comes out of CS and sends a release message to leader so

$K_{\text{free}} = x + y$

Now $x > k$ and $y >= 0$

$\Rightarrow K_{\text{free}} > k$

This is in contradiction with lemma 1 hence our assumption is wrong and theorem is proved.

4.2 Starvation freedom

Lemma 3: Every Request will eventually reach the current leader i.e., token holder node.

Proof: Let us assume the contrary that request of node $i$ never reaches token holder node. A requesting node keep on sending request to new leader whose information is received with leader information message until it has received request acknowledgement. In addition, the outgoing node broadcasts pending requests stored locally to all nodes including token holder node. Moreover, when a node receives a request, it adds it in its local node request list and on receiving token; the new leader transfers the request from local node request list to token request queue, if it is still pending. Therefore, our
assumption is only possible, if the leadership is always transferred to a new node to which the request of node i was never sent before. This is only possible if the number of nodes is infinite which contradicts our system model’s assumption number 8. Hence, our assumption that the request of node i will never reach token holder node is wrong and lemma 3 holds.

**Theorem 2:** Every request will eventually be served.

**Proof:** For a requesting node there are two possibilities.

a) The requesting node is holding the token. This is only possible when there is no pending request and the node is holding idle token. In this case the node will immediately enter the CS.

b) The requesting node doesn’t hold the token.

In this case, the node will send its request to the node which is leader according to its information. This request will eventually reach the token holder node (from lemma 3)

Now there are two possible cases:

1. The token holder is not in CS: In this case, the token will be transferred to the requesting node immediately which will enter CS as leader. Hence the request is served.

2. The token holder is in CS: In this case also there are two possible cases

   2.a The value of $K_{free} > 0$.

   Now in this case the value of $K_{free} > 0$ so the token holder will send a permission granted message and requesting node will enter CS on receiving this message.

   2.b The value of $K_{free} = 0$

   In this case, the request will be added in token queue. Now, each node remains in CS for finite time and the token queue is FCFS. Hence, the request will eventually reach at the head of the token queue and will eventually be served.

From the above discussion it is clear that in all possible cases the request is served. Hence, the theorem is proved.

**4.3 l-deadlock avoidance**

**Lemma 4:** If a non-token holder node comes out of CS, the value of $K_{free}$ will be eventually incremented.

**Proof:** Let us assume the contrary that node $n_i$ has come out of CS, however, the value of $K_{free}$ is never incremented due to this event. On coming out of CS, $n_i$ will send a release message to the node which is the current token holder according to its information. On receiving a release message, token holder node increments the value of $K_{free}$. Hence, our initial assumption is only possible if the release message sent by $n_i$ never reaches token holder node. Since, the release message of $n_i$ has not been received by the token holder node, it will not be deleted from the list of nodes to which permission was sent. Hence, on receiving the new leader(s) information, $n_i$ will find its id in the list of nodes for which release message is still pending and it will keep on sending the release message to new token holder(s). However, on receiving the release message by $n_i$, a node $n_j$ will store the information about the release message of node $n_i$ in its local data structure. Moreover, the outgoing node also broadcasts the pending release information stored locally to all nodes including token holder along with neighborhood leaving information message. Further, when a node receives the token and finds that the information about
release message of \( n_i \) is in its local data structure, it acts as if a release message has been received. Therefore, our initial assumption is true only if the token is transferred to a new node every time before the release message is received by the old token holder. This is only possible if there is infinite number of nodes in the system. However, this contradicts our system model which assumes that there are finite numbers of nodes in the system. Therefore, our initial assumption is invalid and lemma 4 holds.

**Theorem 3:** If there are less than \( k \)-nodes in CS and there are some pending requests then at least one of the requesting nodes will eventually enter CS without waiting for any node to come out of CS.

**Proof:** Since, any pending request will eventually reach token holder node (lemma 3), the request cannot be satisfied only if the value of \( K_{\text{free}} \) is 0. However, since there are less than \( k \) nodes in the critical section, the value of \( K_{\text{free}} \) can be zero only if some nodes which have been allowed by the token holder node have come out of CS but the value of \( K_{\text{free}} \) is not incremented corresponding to their coming out of CS. However, according to lemma 4, if a non-token holder node has come out of CS, value of \( K_{\text{free}} \) will eventually be incremented. Hence, the value of \( K_{\text{free}} \) will become non-zero and some of the requesting nodes will be allowed to enter CS.

5. PERFORMANCE ANALYSIS

In order to analyze resource allocation algorithms in distributed systems, three performance measures [30] namely message complexity, response time and synchronization delay are used. Therefore, in this section, the performance of proposed algorithm has been analyzed using above mentioned performance measures.

5.1 Message complexity

**Message Complexity:** The average number of messages exchanged among the hosts for each execution of the CS.

5.1.1 Message complexity for token holder

According to proposed algorithm, when the token holder node wants to enter CS the value of \( K_{\text{free}} \) will always be a positive value. Therefore, the requesting token holder node can directly enter CS without any message(s) exchanged.

5.1.2 Message complexity for non-token holder (best case)

If a non-token holder node requests CS, the best case will occur when the initial request message of the requesting node reaches to the token holder node before it is transferred to some other node and an instance of resource is free. In that case, the first message is request message followed by one permission grant and one release message. Therefore, the total number of messages required in this case is three.

5.1.3 Message complexity for non-token holder (worst case)

Let the node sends request to token holder and the current token holder transfers the
token to the new token holder node. In this case request is stored at the node from which the token has left and the requesting node has to send request message again to the new token holder. The proposed algorithm ensures that this situation can be repeated at most n-1 times. Therefore, the number of messages required in this case will be n+2 ((n-1) request message followed by one Req_ack, one Per_grant and one Release message).

5.2 Response Time

Response Time: The time interval that a host waits to enter the CS after its request for CS arrives.

Waiting time or response time for a mutual or k-mutual exclusion protocol is generally considered under light load conditions. In this case, the token holder node must be having an idle token with no pending requests. Therefore, the request of any node will reach the current token holder in time T (where T is the maximum message propagation delay). On receiving the request, the current token holder will transfer the token to the requesting node immediately which will reach the requesting node in T time. Hence, the requesting node will receive token and enter CS in 2T time after sending its request message.

5.3 Synchronization Delay

Synchronization Delay: The number of sequential messages exchanged after a host leaves CS and before the next host enters CS.

It is customary to consider synchronization delay under heavy load conditions. In such conditions, the token request queue will never be empty. The best case synchronization delay of KLME is T which happens when the token holder node comes out of CS and immediately transfers the token to the node at the front of the token request queue which in turn enters CS immediately on receiving the token. However, if a non-token holder node comes out of CS, it will first send a release message to the token holder which will select one node from the token queue and will forward a permission-grant message to the selected node. Hence, the synchronization delay in this case will be 2T.

6. SIMULATION OF KLME

6.1 Simulation setup

In order to analyze KLME dynamically, the simulation experiments have been performed using NS2 which is a discrete event driven network simulator. The critical section time of a process is assumed to be exponentially distributed with mean value of 500 milliseconds (μcs). Further, the idle time of a node is also assumed to be exponentially distributed with mean value (μncs). Joung [19] proposed that the contention level can be calculated by using the following formula \( \text{contention level} = \left( \frac{\mu_{cs}}{\mu_{cs} + \mu_{ncs}} \right) \times 100 \). The value of \( \mu_{ncs} \) has been varied to get the desired level of contention. The experiments have been conducted under heavy load (contention level 98%), medium load
(contention level 50%), and light load (contention level 2%) conditions. The number of resources has been varied from 2 to 5. In order to analyze the effect of mobility on the performance of the algorithm, the algorithm has been simulated under light, medium and heavy mobility also.

### 6.1.1 Analysis of simulation results

![Fig. 2(a). Results under light load](image)

![Fig. 2(b). Results under medium load](image)
It is evident from Fig. 2 (a,b,c) that the message complexity of the proposed algorithm varies between n/4 to n/2. The number of messages per CS under heavy load and medium load decreases significantly, however, under light load this difference is not significant. The reason being under light load not enough nodes are requesting which can take advantages of increased resources. As the number of resources are reduced the number of messages exchanged per CS is increased which is a general expected phenomenon. One interesting feature of the results is that number of messages versus number of nodes graph the gradient of plot is very low. Despite varying load the gradients of slope does not show abrupt changes.

When the number of nodes is increased the waiting time also increases under heavy load, however, under medium and light load waiting time is reducing in certain cases on increasing the number of nodes. One possible reason for this phenomenon is that the number of nodes going out of neighborhood and entering the neighborhood also increases. In our algorithm, the outgoing node broadcasts certain information to all other nodes which may help in updating the data structures and information at nodes which may lead to reduced waiting time. When the number of resources is reduced average waiting time is also increases.
Further, the algorithm has been simulated for the different churn rates i.e. for the dynamic environment of MANETs at $k = 5$.

Change in topology is one of the most important features of ad hoc environment. The Fig. 3(a) shows that there is negligible effect of the churn rate on the number of messages/CS even if total number of nodes in the system is increased. In Fig. 3 (b), it is evident that under light churn rate average waiting time is significantly higher than the average waiting time under medium/high churn rate, specially, when the total number of nodes in the system is high. The reason behind this phenomenon is that with low churn rate most of the requesting node will enter CS and with medium/high churn rate a requesting node may move out of neighborhood before entering CS. From the above two graphs, it is clear that the performance of the algorithm does not degrade significantly with increase in churn rate rather it is improving in certain cases.

Fig. 3 (b) clearly indicates that the waiting time stabilizes and does not increase on increasing the number of nodes beyond a limit. The most striking feature of our algorithm is that on increasing the churn rate the waiting time reduces in comparison to low churn rate. This phenomenon can be explained by the fact that a outgoing node broadcasts information captured by it before going out, hence, updating the information on all existing nodes and helps in reducing the waiting time.

7. HANDLING TOKEN LOSS

In token based algorithms, token loss is a major problem. Moreover, this problem further exaggerates because of mobility of nodes in MANETs. Hence, in this section the token loss problem has been handled. For that purpose, following additional assumptions are required.

1. Token acknowledgment will be sent by the new token holder/leader.
2. Old token copy will be maintained by the previous token holder till it receives the token acknowledgement.
3. Token holder node doesn’t fail.
4. Communication channel is assumed to be FIFO.

Detection and Recovery

In the algorithm presented above, token loss may happen when the token is transferred by the current token holder node i to another node j and node j leave the present neighborhood before receiving the token. In this case, leaving node j will broadcast $I_{am\_leaving}$ message. Thus, node i will receive $I_{am\_leaving}$ message from node j. Therefore, node i will detect that the token is lost in transit. Subsequently, it generates new token from the old copy. Now, node i will hold the idle token, in case, the token queue is empty; otherwise, it will transfer the token to the node present at the front of the token queue.

Another possibility is when a token holder node i sends token to some other node j, subsequently, it broadcasts $leader\_info$ message. As message delivery is FIFO, node j must receive token before $leader\_info$ message. However, if node j receives $leader\_info$ before token, node j detects that token is lost. Subsequently, Node j will send $token\_lost$
message to node i. On receiving token_lost message from node j, node i will regenerate the token from its old copy and forward the same to node j.

8. CONCLUSION AND FUTURE SCOPE

The present exposition contains a leader based $k$-local mutual exclusion algorithm for ad hoc networks. $k$-local mutual exclusion is quiet useful in solving several interesting applications for mobile ad hoc networks. The proposed algorithm can be combined for multiple neighborhoods rather than using a conventional KME algorithm specially, when the resources are scattered to multiple locations. To the best of our knowledge, the algorithm proposed in this paper is the first $k$-local mutual exclusion algorithm for MANETs. The algorithm satisfies the correctness needs. It successfully handles link breakage and other changes due to dynamic topology. The static and dynamic analysis of algorithm has been presented. The most striking feature of the algorithm is that the waiting time stabilizes on increasing the number of nodes beyond a limit and decreases with increased churn rate. The token loss which is a major problem in token based algorithms has also been handled. Developing a fault tolerant version of the proposed algorithm may be a future direction of research.

REFERENCES


