A Clustering Group Mutual Exclusion Algorithm
For Mobile Ad Hoc Networks

Jean-Frederic MYOUPO
Gaston Berger University
Laboratoire MIS
University of Picardie Jules Verne
33, rue St-Lieu Amiens Cedex 1 France
Email: myoupo@laria.u-picardie.fr

Mohamed NAIMI
University of Cergy Pontoise
Department of Computer Science
Cergy Pontoise France
Email: naimi@dept-info.u-cergy.fr

Ousmane THIARE
Gaston Berger University
Department of Computer Science
B.P. 234 Saint-Louis Senegal
Email: othiare@dept-info.u-cergy.fr

Abstract—A mobile ad hoc network can be defined as a network that is spontaneously deployed and is independent of any static network. The network consists of mobile nodes with wireless interfaces and has an arbitrary dynamic topology. The networks suffer from frequent link formation and disruption due to the mobility of the nodes. A clustering method is used for obtaining a hierarchical organization for the ad hoc networks. In this paper we present a clustering token based algorithm for Group Mutual Exclusion in ad hoc mobile networks. The proposed algorithm is adapted from the RL algorithm in [1] and utilizes the concept of weight throwing in [2]. The proposed algorithm is sensitive to link forming and link breaking. The algorithm ensures the mutual exclusion, the bounded delay, and the concurrent entering properties.

Keywords: Critical section, mutual exclusion, group mutual exclusion, clustering, ad hoc networks.

I. INTRODUCTION

A mobile ad hoc network can be defined as a network that is spontaneously deployed and is independent of any static network. The network consists of mobile nodes with wireless interfaces and has an arbitrary dynamic topology.

Wireless links failure occur when nodes move so that they are no longer within transmission range of each other. Likewise, wireless link formation occurs when nodes move so that they are again within transmission range of each other. In [1], an algorithm is proposed to solve the mutual exclusion problem for mobile ad hoc networks. The mutual exclusion problem is concerned with how to control nodes to enter the critical section to access a shared resource in a mutually exclusive way. The Group Mutual Exclusion (GME) is a generalization of the mutual exclusion problem. In the GME problem, multiple resources are shared among nodes. Nodes request to access the same shared resource may do so concurrently. However, if nodes compete to access different resources, only one of them can proceed.

In addition to the paper [1], there are methods proposed to solve mutual exclusion related problems for ad hoc networks. The paper [3] is proposed for solving the k-mutual exclusion problem, [5], for the leader election problem. There are several papers proposed to solve the GME problem for different system models. The papers [6][4] are designed for distributed message passing models, the paper [9], for self-stabilizing models.

We adapt the solution of [1] to solve the GME problem for mobile ad hoc networks. We utilize also the concept used in [2] to detect that all the nodes concurrently accessing the same resource have terminated their tasks (concurrent entering). In [11], R. Mellier et J-F. Myoup have presented a MUTEX protocol for multi-hop MANETs which takes advantages of the cluster structure offered by the partitioning techniques.

This paper is organized as follows: the next section discusses related work. Section III presents some preliminaries and our algorithm. We prove the algorithm correctness in section IV. Conclusion and future work are offered in section V.

II. RELATED WORK

In [1], a token-based mutual exclusion algorithm, named RL (Reverse Link), for ad hoc networks is proposed. We takes the same assumptions on the mobile nodes and network like in [1].

The RL algorithm also assumes that there is a unique token initially and utilizes the partial reversal technique in [8] to maintain a token oriented DAG (directed acyclic graph). In the RL algorithm, when a node wishes to access the shared resource, it sends a request message along one of the communication link. Each node maintains a queue containing the identifiers of neighborings nodes from which it has received request for the token.

Now we present the scenario for the GME problem. Consider an ad hoc network consisting of n nodes and m shared resources. Nodes are assumed to cycle through a non-critical section (NCS), an waiting section (Trying), and a critical section (CS). A node i can access the shared resource only within the critical section. Every time when node i wishes to access a shared resource $S_i$, node i moves from its NCS to the Trying, waiting for entering the CS. The GME problem [7] is concerned with how to design an algorithm satisfying the following property:

- Mutual Exclusion: If two distinct nodes, say i and j, are in the CS simultaneously, then $S_i = S_j$.

1 The terms processes and nodes will be used interchangeably throughout the paper.
• **Bounded Delay:** If a node enters the Trying protocol, then it eventually enters the CS.

• **Concurrent Entering:** If there are some nodes requesting to access the same resource while no node is accessing a different resource, then all the requesting nodes can enter CS concurrently.

### III. PROPOSED ALGORITHM

#### A. Preliminaries

Every node $i$ in the networks is assigned a unique identifier (ID). For simplicity, here we identify each node with its ID and we denote both with $v$. Finally, we consider weighted networks, i.e., a weight $w_i$ (a real number $\geq 0$) is assigned to each node $i_{0\leq i\leq n-1}$ of the network.

Clustering in ad hoc network means partitioning nodes into clusters, each one with a clusterhead and (possibly) some ordinary nodes. The choice of the clusterheads is here based on the weight associated to each node. The bigger the weight of the node, the better that node for the role of clusterhead. In order to meet the requirements imposed by the wireless, mobile nature of these networks, a clustering algorithm is required to partition the nodes of the network so that the following ad hoc clustering properties are satisfied:

1. Every ordinary node has at least a clusterhead as neighbor (dominance property).
2. Every ordinary node affiliates with the neighboring clusterhead that has the bigger weight.
3. No two clusterheads can be neighbors (independence property).

Property 1. is necessary to ensure that each ordinary node has direct access to at least one clusterhead (the one of the cluster to which it belongs), thus allowing fast intra- and inter-cluster communications. The second property ensures that each ordinary node always stays with the neighboring clusterhead with the bigger weight, i.e., with the clusterhead that can give it a "guaranteed good" service. Finally, property 3. guarantees that the network is covered by a "well scattered" set of clusterheads, so that each node in the networks has a clusterhead in its neighborhood and it has direct access to that clusterhead.

For the sake of simplicity, we stipulate that each node has a different weight. As an example, the topology of a simple ad hoc network is shown in Figure 1.

![Figure 1](image1.png)

**Figure 1.** (a) An ad hoc network $G$ with nodes $v$ and their weights ($w_v$), $1 \leq v \leq 8$, and (b) a correct clustering for $G$.

#### B. Algorithm

In this algorithm, we assume that all the nodes concurrently accessing the same resource terminate their tasks. The algorithm is assumed to execute in a system consisting of $n$ nodes and $m$ shared resources. Nodes are labeled as $0, 1, \cdot \cdot \cdot, n-1$, and resources are labeled as $0, 1, \cdot \cdot \cdot, m-1$. We assume there is a unique token held by node $0$ initially. Variables used in the algorithm by node $i$.

• **status:** indicates whether node is the Trying, CS, or NCS. Initially, $status=\text{NCS}$.

![Figure 2](image2.png)

**Figure 2.** States processes

- $N$: the set of all nodes in direct wireless contact with node $i$. Initially, $N$ contains all of node $i$'s neighbors.
- $w_i$: the weight associated to node $i$.
- **weight:** a variable used for weight throwing. Initially, weight is set to $0$ for every node.
- **Cluster(i):** the set of nodes in $i$'s cluster. It is initialized to $\emptyset$.
- **height:** a three-tuple $(h_1, h_2, i)$ representing the height of node $i$.
- **Vect:** an array of tuples representing node $i$’s view of height of node $i$, $i \in N$. Initially, Vect[i]=height of node $i$.
- **Leader:** a flag set to true if node holds the token and set to false otherwise. If Leader=true then the node i is a clusterhead. Initially, Leader=true if $i=0$, and Leader=false otherwise.
- **next:** indicates the location of the token from node i’s viewpoint.
- **Q:** a queue which contains request of neighbors.
- **receivedLI:** boolean array indicating whether the height carrying message LinkInfo has been received from node j, to which a Token() message was recently sent.
- **forming[j]:** boolean array set to true when link to node $j$ has been detected as forming and reset to false when first LinkInfo message arrives from node j. Initially, forming[j]=false for all \( j \in N \).
- **formHeight[j]:** an array storing the value of height of node $j$.

Messages used in the algorithm:

- **Request():** when a node i wishes to enter the CS to access the resource S, it sends out Request() to the neighbor node indicated by next.
- **SubToken():** a message to inform nodes to access the resource S concurrently. There may be several subtokens in the system simultaneously.
• **Token()**: a message for node to enter the CS. The node with token is called the **Leader**.
• **Rel()**: a message for node i to release the resource **S**<sub>i</sub>, it sends out **Rel()** to one of the neighbor node.
• **LinkInfo()**: a message used for nodes to exchange their height values with neighbors.

The principle of algorithm is the following:

We assume that the DCA (Distributed Clustering Algorithm) [10] protocol has set up a cluster structure over the network and that, now the network is mobile according the restriction that the structure of the cluster must not change.

There is a single token on the network and we assume that the token is safe and reliable. The token passes from a **clusterhead** to another. The nodes that are not clusterheads just play the role of router. When a **clusterhead** holds the token, it sends successively a **SubToken()** message to all its neighbors (the nodes found in the **Cluster(i)**). Each of them gets the CS if it wants and then sends back the **SubToken()** message to its clusterhead. The **clusterhead** who holds the token also informs other clusterheads of the opened session. The nodes of other clusters who wish to enter this session do after being informed by their clusterhead. This is made possible with the **dominance property**. Once all its affiliated neighbors, including itself, have been satisfied, it sends the token to the next clusterhead. In our algorithm, we consider the next clusterhead as one whose size of the cluster is the largest in terms of numbers of nodes. It may happen that the remaining clusters are the same size, in this case we choose the **clusterhead** who is in **Cluster()** who has the greater ID.

Like the RL algorithm, the proposed algorithm is event-driven. An event at node i consists of receiving a message from another node, or an indication of link failure or formation from the link layer, etc. Each event triggers a procedure which is assumed to be executed atomically. Below, we present the overview of the event-driven procedures:

• **Requesting the resource R**: When node i requests to enter the CS to access resource R, it enqueues the message **Request()** on Q and sets status to **Trying**. If node i does not currently hold the token, it calls **SendRequest()** to send a request message. If node holds the token that means that i is a **clusterhead**, i then sets weight to 0, removes **Request()** from Q and sets status to **CS** to access resource R, since its request will be at the head of Q. Node i also sends **SubToken()** to all neighbors which are in **Cluster(i)**.

• **Releasing the resource R**: When a non-token holding nodes i leaves the CS to release resource R, it calls **SendRel()** to send out **Rel(w)** message with w=weight, to one of the neighbors and sets status to NCS. Node i then calls **SendTokenToNext()** to send the token to the next clusterhead and sets status=NCS.

2The terms session and resource will be used interchangeably throughout the paper

• **Receiving a request message**: When a **Req(j,R)** message sent by a neighbor node j is received at node i, i ignores the request message if receivedLink[j] is false. Otherwise, i changes **Vecr(j)** and enqueues the request on Q if the link between i and j is incoming at i. If Q is not empty, and status=NCS, i calls **SendTokenToNext()** provides i holds the token.

• **Receiving a release message**: Suppose that node i is a **clusterhead** and holds the token, then when a **Rel(w)** message sent by a neighboring node j is received at node i, i decreases weight by w and checks if weight is 0 and status is NCS. If so, it means that all nodes accessing the same resource have completed their tasks.

• **Receiving the token message**: When node i receives a token message from some neighbor j, i set leader to true. Then i lowers its height to be lower than that of the last token holder, node j, informs all its neighbors of its new height by sending **LinkInfo()** messages, and calls **SendTokenToNext()**.

• **Receiving a SubToken() message**: When node i receives an **SubToken(R,w)** message from some neighbor j, i splits w into **w_1, w_2, · · · , w_q, q = |Q|, fractions**. Node i then sends **SubToken(R, w_1)**, **SubToken(R, w_2)**, · · · , **SubToken(R, w_q)**, respectively, to the q neighbors whose requests are in Q. If i’s request message for accessing resource R is in Q, i can enter the CS and access the resource R. In this case, node i sets weight = **w_i**, where **w_i** is the fraction of w attached in **SubToken()** send for the i’s request in Q. Moreover, if i’s request message **Request(i,S)** is the only request in Q and S ≠ R, then i sends out the **Rel(w)** message by calling **SendRel()**. Note that all the request messages for accessing resource R will be deleted from Q in this event handling procedure.

• **Receiving a link information message**: When a link information message **LinkInfo()** from node j is received at node i, j’s height is recorded in **Vect[j]**. If **receiveLink[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 **receivedLink[j]** to true. If **forming[j]** is true, the current value of height is compared to the value of height when the link to j was first detected, **formHeight[j]**.

• **Link failing**: When node i senses the failure of a link to a neighboring node j, it removes j from N, sets **receivedLink[j]** to true, and if j is an element of Q, deletes j from Q.

• **Link forming**: When node i detects a new link to node j, i sends a **LinkInfo()** message to j, sets **forming[j]** to true, and sets **formingHeight[j] = height**.

The following are some procedures called by the event handling procedures introduced above.

• **Procedure SendTokenToNext()**: Node i dequeues the first request, say **Request(j,S)**, on Q and sets next to j. If next=i, i enters the CS. After i enters the CS, node i sends q, q = |Q|, **SubToken(R,w)** with w=1 to neighbors whose requests in Q. Since each **SubToken** a weight value to 1,
node $i$ then increase weight by $q$ and remove the request messages for accessing resource $R$ from $Q$.

- **Procedure `raiseHeight()`**: Called at non-token holding node $i$ when $i$ loses its last outgoing link. Node $i$ raises its height using the partial reversal method of [8] and informs all its neighbors and informs all its neighbors of its height change with `LinkInfo()` message.

- **Procedure `SendRequest()`**: Selects nodes $i$’s lowest-height neighbor to be $\text{next}$. Send a message to $\text{next}$.

- **Procedure `SendRel()`**: A non-token holding node $i$ calls `raiseHeight()` when $i$ loses its last outgoing link. After calling `raiseHeight()`, selects its lowest-height neighbor to be $\text{next}$ and sends a release message to $\text{next}$. The `SendRel()` procedure is never called by a token-holding node.

### IV. PROOF OF THE ALGORITHM

In this section we prove that the algorithm satisfies the following three properties: mutual exclusion between sessions, the bounded delay, and the concurrent entering.

**Theorem 1**: The algorithm ensures the mutual exclusion between sessions.

**Proof 2**: When a node $i$ is a clusterhead and holds the token, it can enter the CS and then sends out subtokens to requesting neighbors which are in $\text{Cluster}(i)$. When a node receives a subtoken, it can enter the CS if it request for the same resource as the the token holder. Since there is only a unique token, all nodes in CS must access the same resource. Thus, the mutual exclusion property is guaranteed.

Below, we show that the proposed algorithm satisfies the concurrent entering property in Theorem 3. When a node $i$ is a clusterhead and holds the token, it can enter the CS and then sends out subtokens to requesting neighbors which are in $\text{Cluster}(i)$. When a node receives a subtoken, it can enter the CS if it request for the same resource as the the token holder. To sum up, all nodes can access the same resource as the token holder is currently accessing. So, the concurrent entering property is guaranteed.

### V. CONCLUSION AND FUTURE WORKS

We presented a token-based algorithm to solve the group mutual exclusion for mobile ad hoc networks. This algorithm takes advantages of the cluster structure offered by the partitioning techniques of [10]. The algorithm is adapted from the RL algorithm in [1] and utilizes a concept used in [2] to detect that all nodes concurrently accessing the same resource have terminated their tasks. The algorithm reduces the message complexity. Simulations and message complexity is left as a future task. As [11], the problem of energy efficient for this algorithm remains open.

### REFERENCES


