6th International Conference on Computer Science and Informatics, pp. 266-270, March 2002

A Group Mutual Exclusion Algorithm for Ad Hoc Mobile Networks

Jehn-Ruey Jiang

Information Management Department
Hsuan Chuang University
HsinChu, 300, Taiwan

Abstract

In this paper, we propose a token based algorithm to solve the group mutual exclusion (GME) problem for ad hoc mobile networks. The proposed algorithm is adapted from the RL algorithm in [WWV98] and utilizes the concept of weight throwing in [Tse95]. We prove that the proposed algorithm satisfies the mutual exclusion, the bounded delay, and the concurrent entering properties. The proposed algorithm is sensitive to link forming and link breaking and thus is suitable for ad hoc mobile networks.

Keywords: *mutual exclusion, group mutual exclusion, ad hoc networks, distributed algorithms*

1. Introduction

In this paper, we propose a token-based algorithm to solve the group mutual exclusion (GME) problem for ad hoc mobile networks. An ad hoc mobile network consists of mobile nodes which can communicate with each other by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes. Wireless link "failures" 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 paper [WWV98], an algorithm is proposed to solve the mutual exclusion problem for ad hoc mobile 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) problem is a generalization of the mutual exclusion problem. In the GME problem, multiple resources are shared among nodes. Nodes requesting 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 [WWV98], there are papers proposed to solve mutual exclusion related problems for ad hoc networks. The paper [WCM01] is proposed for solving the *k*-mutual exclusion problem, the paper [MWV00], for the leader election problem. There are several papers proposed to solve the GME problem for different system models. The papers [Jou99, WJ99, CDPV01] are designed for distributed message passing models, the papers [Jou98, KM99], for shared memory models, and the paper [CP00], for self-stabilizing models. In this paper, we adapt the solution of [WWV98] to solve the group mutual exclusion problem for ad hoc mobile networks. Note that we also apply the weight-throwing concept in [Tse95] to detect that all the nodes concurrently accessing the same resource have terminated their tasks.

The rest of this paper is organized as follows. In section 2, we introduce some preliminaries. We introduce the proposed algorithm in section 3, and prove the algorithm correctness in section 4. At last, we give a concluding remark in section 5.

2. Preliminaries

In [WWV98], a token-based mutual exclusion algorithm, named RL (Reverse Link), for ad hoc networks is proposed. The RL algorithm takes the following 7 assumptions, which we also take in this paper.

1. The nodes have unique node identifiers.
2. Node failures do not occur.
3. Communication links are bidirectional and FIFO.
4. 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.

5. Incipient link failures are detectable.
6. Message delays obey the triangle inequality (*i*.e., messages that travel 1 hop will be received before messages sent at the same time that travel more than 1 hop).
7. Partitions of the network do not occur.

The RL algorithm also assumes that there is a unique token initially and utilizes the partial reversal technique in [GB81] 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 neighboring nodes from which it has received requests for the token. The RL algorithm totally orders nodes so that the lowest-ordered node is always the token holder. Each node dynamically chooses its lowest-ordered neighbor as its outgoing link to the token holder. Nodes sense link changes of immediate neighbors and reroute requests based on the order newly created. The token holder grants the token according to the requests' positions in the queue, and thus requests are eventually served while the DAG is being re-oriented and blocked requests are being rerouted.

Now we present the scenario for the group mutual exclusion 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 entry section (*ES*), and a critical section (*CS*). A node i can access the shared resource only within the critical section. Every time when a node i wishes to access a shared resource R_i, node i moves from its *NCS* to the *ES*, waiting for entering the *CS*. The Group Mutual Exclusion (GME) problem [Jou98] 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 $R_i = R_j$.

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 the *CS* concurrently.

Bounded Delay: If a node enters the *ES*, then it eventually enters the *CS*.

3. Proposed Algorithm

In this section, we propose a distributed algorithm to solve the group mutual exclusion (GME) problem for ad hoc mobile networks. The algorithm is assumed to execute in a system consisting of n nodes and m shared resources. Nodes are labeled as 0, 1, …, n-1, and resources are labeled as 0, 1, …, m-1. We assume there is a unique token held by node 0 initially. The variables used in the algorithm for node i are listed below. Note that subscripts are included when needed.

- *state*: Indicates whether node i is in the *ES*, *CS*, or *NCS* state. Initially, *state* = *NCS*.
- N: The set of all nodes in direct wireless contact with node i. Initially, N contains all neighbors of node i.
- *height*: A triplet (h_1, h_2, i) representing the height of node i. Links are considered to be directed from higher-height nodes toward lower-height nodes, based on lexicographic ordering. For example, if the height of node 1, $height_1$, is (2, 3, 1) and the height of node 2, $height_2$, is (2, 2, 2), then $height_1 > height_2$ and the link would be directed from node 1 to node 2. Initially, $height_0 = (0, 0, 0)$, and $height_j$, $j \neq 0$, is initialized so that the directed links form a DAG where each node has a directed path to node 0.
- *hVector*: An array of triplets representing node i's view of *height* of node j, $j \in N$. Initially, *hVector*[j] = *height* of node j. From node i's viewpoint, the link between i and j is incoming to node i if *hVector*[j] $> height_i$, and outgoing from node i if *hVector*[j] < $height_i$.
- *leader*: A flag set to true if node i holds the token and set to false otherwise. Initially, *leader* = true if $i = 0$, and *leader* = false otherwise.
- *next*: Indicates the location of the token from node i's viewpoint. 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.
- *weight*: a variable used for weight throwing. Initially, *weight* is set to 0 for every node.
- Q: a queue which contains requests of 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, $Q = \varnothing$.
- *receivedLINK*[j]: Booloen array indicating whether the height carrying message LINK has been received

from node *j*, to which a TOKEN message was recently sent. Any height information received at node *i* from node *j* for which *receivedLINK*[*j*] is false will not be recorded in *hVector*[*j*]. Initially, *receivedLINK*[*j*]=true for all $j \in N$.

- *forming*[*j*]: Boolean array set to true when link to node *j* has been detected as forming and reset to false when first LINK message arrives form node *j*. Initially, *forming*[*j*]=false for all $j \in N$.
- *formHeight*[*j*]: An array storing the value of *height* of node *j*, $j \in N$, when new link to *j* is first detected. Initially, *formHeight*[*j*]=*height* for all $j \in N$.

The following are the messages used in the algorithm. Note that each message is attached with the *height* value, denoted by *h*, of the node sending the message.

- REQUEST(*i*, *R*): When *i* wishes to enter the *CS* to access the resource *R*, it sends out REQUEST(*i*, *R*) to the neighbor node indicated by *next*.
- RELEASE(*w*): When *i* leaves the *CS* to release the resource *R*, it sends out RELEASE(*w*) to one of the neighbor node.
- TOKEN: a unique message for node to enter the *CS*. The node with TOKEN is called the leader.
- SUBTOKEN(*R*, *w*): a message to inform nodes to access the resource *R* concurrently with weight *w*. Note that there may be several SUBTOKENs in the system simultaneously.
- LINK: a message used for nodes to exchange their height values with neighbors.

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(*i*, *R*) on *Q* and sets *state* to *ES*. 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(*i*, *R*) message. If node *i* does hold the token, *i* then sets *weight* to 0, removes REQUEST(*i*, *R*) from *Q* and sets *state* to *CS* to access resource *R*, since its request will be at the head of *Q*. If node *i* receives any request message, say REQUEST(*j*, *S*), while it is in the *CS*, it sends SUBTOKEN(*R*, *w*) with *w*=1 to every requesting neighbor. For each SUBTOKEN sent, node *i* increments *weight* by 1. Note that the request message REQUEST(*j*, *S*) is enqueued on *Q* if $S \neq R$. If *S*=*R*, then REQUEST(*j*, *S*) is fulfilled with SUBTOKEN(*R*, *w*), it does not need to be enqueued on *Q*.
- Releasing the resource *R*: When a non-token holding node *i* leaves the *CS* to release resource *R*, it calls *sendRelease*() to send out RELEASE(*w*) message, with *w*=*weight*, to one of the neighbor and sets *state* to *NCS*. If *i* is the token holder, *i* checks if *weight*=0. If so, it means that all nodes accessing the same resource have completed their tasks. Node *i* then calls *transmitToken*() and sets *state*=*NCS*.
- Receiving a request message: When a REQUEST(*j*, *R*) message sent by a neighboring node *j* is received at node *i*, *i* ignores the request message if *receivedLink*[*j*] is false. Otherwise, *i* changes *hVector*[*j*] and enqueues the request on *Q* if the link between *i* and *j* is incoming at *i*. If *Q* is non-empty, and *state* = *NCS*, *i* calls *transmitToken*() 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.
- Receiving a release message: Suppose node *i* holds the token, then when a RELEASE(*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 *state* is *NCS*. If so, it means that all nodes accessing the same resource have completed their tasks. Thus, *i* calls *transmitToken*() to pass the token. Suppose node *i* does not hold the token, then when *i* receives a RELEASE(*w*) message, *i* just calls *sendRelease*() to forward the release message.
- Receiving the token message: When node *i* receives a TOKEN message from some neighbor *j*, *i* sets *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 LINK messages, and calls *transmitToken*().
- Receiving a subtoken message: When node *i* receives a SUBTOKEN(*R*, *w*) message from some neighbor *j*, *i* splits *w* into $w_1, w_2, \ldots, w_q$, $q=|Q|$, 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 $S \neq R$, then i sends out the RELEASE(w) message by calling *sendRelease*(). 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 LINK from node j is received at node i, j's height is recorded in $hVector[j]$. If $receivedLINK[j]$ 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, $formHeight[j]$. if *height* and $formHeight[j]$ are different, then a LINK 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 a node of 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 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. Then, if i is not the token holder and i has no outgoing links, i calls *raiseHeight*(). If node i is not the token holder, Q is non-empty, and the link to *next* has failed, i calls *forwardRequest*() since it must send another request message for the token.
- Link forming: When node i detects a new link to node j, i sends a LINK message to j, sets $forming[j]$ to ture, and sets $formingHeight[j]=height$.

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

- Procedure *transmitToken*(): Node i dequeues the first request, say REQUEST(j, S), on Q and sets *next* equal to j. If $next = i$, i enters the *CS*. After i enters the *CS*, node i sends q, $q=|Q|$, SUBTOKEN(R, w)s with w=1 to neighbors whose requests in Q. Since each SUBTOKEN carries a weight value of 1, node i then increases *weight* by q and remove the request messages for accessing resource R from Q. If $next \neq i$, i lowers $hVector[next]$ to ($height.h_1$, $height.h_2 - 1$, *next*), so any incoming REQUEST message will be sent to *next*, sets *leader* to false, sets $receivedLINK[next]$ 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 raises its height using the partial reversal method of [GB81] and informs all its neighbors of its height change with LINK 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 message for the token.
- Procedure *forwardRequest*(): Selects node i's lowest-height neighbor to be *next*. Sends a request message to *next*.
- Procedure *sendRelease*(): A non-token holding node i calls *raiseHeight*() when i loses it's last outgoing link. After calling *raiseHeight*(), i selects its lowest-height neighbor to be *next* and sends a release message to *next*. The *sendRelease*() procedure is never called by a token-holding node.

4. Correctness

In this section, we prove that the proposed algorithm satisfies the mutual exclusion, the concurrent entering, and the bounded delay properties. We first show that the proposed algorithm satisfies the mutual exclusion property in Theorem 1.

Theorem 1. The algorithm ensures the mutual exclusion property.

Proof:

When a node owns the token, it can enter the *CS* and then it sends out subtokens to requesting neighbors. When a node receives a subtoken, it can enter the *CS* if it requests for the same resource as the token holder. Since there is only a unique token, all nodes in the *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 2.

Theorem 2. The algorithm ensures the concurrent entering property.

Proof:

When a node owns the token, it can enter the *CS* and then it sends out subtokens to requesting neighbors. When a node receives a subtoken, it can enter the *CS* if it requests for the same resource as the token holder. To sum up, all nodes can access the same resource as that the token holder is currently accessing. So, the concurrent entering property is guaranteed. ■

Below, we prove that the proposed algorithm satisfies the bounded delay property by showing that a requesting node owns the token eventually. Since the height values of the nodes are totally ordered, the logical graph whose arcs are assumed to have the direction from higher *height* values to lower *height* values cannot have any cycles, and thus it is a DAG (Directed Acyclic Graph). We want to show that the DAG is token oriented, i.e., for every node *i*, there exists a directed path originating at node *i* and terminating at the token holder. We present Lemma 1, which is the very Lemma 4 of [WWV98].

Lemma 1. If link changes cease, the logical graph, whose arcs are assumed to have the direction from higher *height* values to lower *height* values, is a token oriented DAG. ■

On the basis of Lemma 1, we can prove that a requesting node owns token eventually.

Theorem 3. If link changes cease, then a requesting node owns the token eventually.

Proof:

When a token holder *i* is in the *NCS* with *weight*=0, it passes the token to the node *j* whose request is at the head of the queue. Node *i* then removes *j*'s request from the queue after passing the token. So, every node's request will eventually be at the head of the queue to have the opportunity to own the token. By Lemma 1, every node's request has a path leading to the token holder. So, a requesting node owns the token eventually. ■

5. Concluding Remarks

In this paper, we have proposed a token based algorithm to solve the group mutual exclusion (GME) problem for ad hoc mobile networks. The proposed algorithm is adapted from the RL algorithm in [WWV98] and utilizes the concept of weight throwing in [Tse95]. We have proved that the proposed algorithm satisfies the mutual exclusion, the bounded delay, and the concurrent entering properties. The proposed algorithm is sensitive to link forming and link breaking and thus is suitable for ad hoc mobile networks.

References

[CDPV01] S. Cantarell, A. K. Datta, F. Petit, and V. Villain. Token based group mutual exclusion for asynchronous rings. *21th International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 691-694, 2001.

[CP00] S Cantarell and F Petit. Self-Stabilizing Group Mutual Exclusion for Asynchronous Rings. *4th International Conference On Principles Of DIstributed Systems (OPODIS 2000)*, pages. 71-90, 2000.

[GB81] E. Gafni and D. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology, *IEEE Transactions on Communications*, C-29(1):11-18, 1981.

[Jou98] *Y.*-J. Joung. Asynchronous group mutual exclusion (extended abstract). *17th Annual ACM Symposium on Principles of Distributed Computing (PDOC)*, pages 51-60, 1998.

[Jou99] *Y.*-J. Joung. The congenial talking philosophers problem in computer networks (extended abstract). *13th International Symposium on DIStributed Computing (DISC'99)*, 1999.

[KM99] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 23-32. ACM Press, 1999.

[MWV00] N. Malpani, J. L. Welch, and N. H. Vaidya, Leader Election Algorithms for Mobile Ad Hoc Networks, *Proc. Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pp. 96-103, 2000.

[Tse95] Y.-C. Tseng, "Detecting Termination by Weight-Throwing in a Faulty Distributed System," *Journal of Parallel and Distributed Computing*, vol. 25, 1995, pp. 7-15.

[WCM01] J. Walter, G. Cao, and M. Mohanty, A *k*-Mutual Exclusion Algorithm for Ad Hoc Wireless Networks, *Proceedings of the first annual Workshop on Principles of Mobile Computing (POMC 2001)*, August, 2001.

[WJ99] K.-P. Wu and *Y.*-J. Joung. Asynchronous group mutual exclusion in ring networks. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP '99)*, pages 539-543, 1999.

[WWV98] J. Walter, J. Welch, and N. Vaidya, A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks, *1998 Dial M for Mobility workshop*, Dallas TX, 1998, 15 pgs.