PRIORITIZED *h*-OUT OF-*k* MUTUAL EXCLUSION FOR MOBILE AD HOC NETWORKS AND DISTRIBUTED SYSTEMS

Jehn-Ruey Jiang
Department of Information Management
Hsuan Chuang University
Hsin-Chu, 300, Taiwan
E-mail:jrjiang@hcu.edu.tw

Abstract

In this paper, we propose a distributed *prioritized h-out of-k mutual exclusion* algorithm for a *mobile ad hoc network* (*MANET*) with real-time or prioritized applications. The *h*-out of-*k* mutual exclusion problem is a generalization of the *k-mutual exclusion* problem and the *mutual exclusion* problem. The proposed algorithm is sensitive to link forming and link breaking and thus is suitable for a MANET. It is worthwhile to mention that the proposed algorithm can also be applied to distributed systems consisting of stationary nodes that communicate with each other by exchanging messages over wired links.

Key Words

mobile ad hoc network (*MANET*), *distributed systems*, *mutual exclusion*, *resource allocation*, *real-time systems*, *prioritized systems*

1. Introduction

In this paper, we propose a distributed *prioritized h-out of-k mutual exclusion* algorithm for a *mobile ad hoc network* (*MANET*). A MANET [18] consists of mobile nodes that 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. It is worthwhile to mention that the proposed algorithm can also be applied to distributed systems consisting of stationary nodes that communicate with each other by exchanging messages over wired links.

Consider a MANET with *k* identical shared resources. Assume that a node in the MANET has to occasionally access *h* out of the *k* shared resources to complete its job. The problem of controlling the nodes so that each node can acquire the desired number of resources with the restriction that the total number of resources simultaneously accessed by nodes should not exceed *k* is called the *h-out of-k mutual exclusion* problem or the *h-out of-k resource allocation* problem [23]. The *h*-out of-*k* mutual exclusion problem is a generalization of the *k-mutual exclusion* problem [1] and the *mutual exclusion* problem [5]. If we restrict *h* to be 1, then the *h*-out of-*k* mutual exclusion problem becomes the *k*-mutual exclusion problem, in which at most *k* nodes are allowed to concurrently access one shared resource. If we restrict both *h* and *k* to be 1, then *h*-out of-*k* mutual exclusion problem becomes the mutual exclusion problem, in which only one node at a time is allowed to access the sole shared resource.

In handheld CSCW (Computer Supported Cooperative Work) environment [25], we can see the application of *h*-out of-*k* mutual exclusion; we can rely on it to achieve coordination among cooperative nodes. For example, consider an electronic whiteboard system, which has three permissions for nodes to draw simultaneously. We can apply *h*-out of-3 mutual exclusion to control such a system by taking permissions as resources. Generally, a node should request for one resource (permission) to draw. Once for a while, a node may wish to be the sole one that draws. In such a case, the node should obtain all the three permissions to draw. Recently, some researches start to discuss the cooperative work environment for MANET. For example, cooperative tablet computer system [19], cooperative robotics and nanorobotics [4], IPAD (inter-personal awareness devices) cooperative work (including Hummingbirds System [10], generalized Hummingbirds System [24], Hocman System [6], etc). When devices need coordination, *h*-out of-*k* mutual exclusion may be applied to the systems just mentioned.

In the *h*-out of-*k* mutual exclusion problem, nodes access the shared resource in the "*first come first serve* (*FCFS*)" manner; however, in the prioritized *h*-out of-*k* mutual exclusion problem, nodes access the shared resource in the "*highest priority first serve* (*HPFS*)" manner. The HPFS criterion arises in real time systems where the tasks have to meet deadlines; it also arises in prioritized systems where key tasks must proceed quickly for good performance. In real time systems, the node with the task of the earliest deadline is assumed to have the highest priority; while in the prioritized systems, the node with the most significant task is assumed to have the highest priority.

There are several distributed prioritized mutual exclusion algorithms [2, 3, 8, 9, 11, 12, 13, 21, 22, 26] proposed in the literature. There are also several algorithms [16, 17, 20] proposed to solve the *h*-out of-*k* mutual exclusion problem for distributed systems. One possible way to provide mutual exclusion related primitives for MANETs is to execute the existent distributed algorithms on top of routing protocols, as depicted in Fig. 1. The other way to provide the mutual exclusion related primitives is to consider the essence of the primitives and dose not rely on any routing protocols (refer to Fig. 2). Some mutual exclusion algorithms for MANETs [14, 15, 27] take this approach.

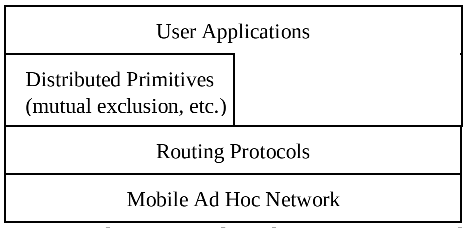

Figure 1. Providing mutual exclusion primitives based on routing protocols for MANETs.

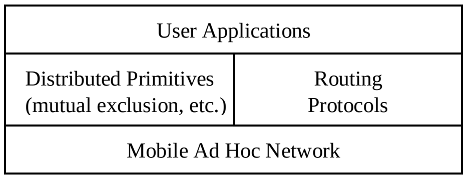

Figure 2. Providing mutual exclusion primitives not based on routing protocols for MANETs.

With the structure of Fig. 2, Jiang proposed a token-based algorithm to solve the *h*-out of-*k* mutual exclusion problem for MANETs in [15]. Jiang's algorithm applies the *RL* (*Reverse Link*) technique to maintain a token oriented DAG (directed acyclic graph). A node should gain the token along the DAG to access the shared resource. The RL technique endows Jiang's algorithm with the ability of being sensitive to link forming and link breaking. This is why Jiang's algorithm can be applied to MANETs.

In this paper, we utilize the concept of RL to implement prioritized *h*-out of-*k* mutual exclusion algorithm for MANETs. In addition to the concept of the RL technique, we also utilize the concept of *priority queue* and *priority update* to endow the algorithm with the ability of HPFS property. Furthermore, we adopt the concept of *aging* to prevent a node from being always preempted by nodes with higher priorities. Thus, the proposed algorithm is *starvation-free* and can be properly applied to MANETs with real-time or prioritized applications.

The rest of this paper is organized as follows. In section 2, we introduce some preliminaries. We present 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 [27], a token-based mutual exclusion algorithm, named RL (Reverse Link), for a MANET is proposed. The RL algorithm takes the following 6 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. 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 [7] 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 prioritized *h*-out of-*k* mutual exclusion problem. Consider a MANET consisting of *n* nodes and *k* 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 a node *i* wishes to access *h* shared resources, node *i* moves from its *NCS* to the *ES*, waiting for entering the *CS*. The prioritized *h*-out of-*k* mutual exclusion problem is concerned with how to design an algorithm satisfying the following properties:

Mutual Exclusion:
No more than *k* resources can be accessed concurrently.

Highest Priority First Serve:
If there are nodes competing for entering the *CS*, the node with the highest priority will proceed first.

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

3. The Proposed Solution

In this section, we propose a distributed token-based algorithm to solve the prioritized *h*-out of-*k* mutual exclusion problem for a MANET. The algorithm is assumed to execute in a system consisting of *n* nodes and *k* shared resources. Nodes are labeled as 0, 1, …, *n*-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 the subscript "*i*" is 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 (neighbors) 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.
- *htVector*: An array of triplets representing node *i*'s view of *height* of node *j*, $j \in N$. Initially, *htVector*[*j*] = *height* of node *j*. From node *i*'s viewpoint, the link between *i* and *j* is incoming to node *i* if *htVector*[*j*] >$height_i$, and outgoing from node *i* if *htVector*[*j*] < $height_i$.
- *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.
- *tokenHolder*: a boolean variable indicating whether or not node *i* holds the token. If node *i* holds the token, *tokenHolder* is set to true. It is set to false, otherwise.
- *Q*: a queue which contains requests of neighbors. Initially, *Q* = ∅. Operations on *Q* include *enqueue*, *dequeue*, and *delete*. The *enqueue* operation inserts an item at the rear of *Q*, and the *dequeue* operation returns and removes the item at the front of *Q*, and the *delete* operation removes a specified item from *Q*, regardless of its location.
- *receivedLink*[*j*]: a boolean array indicating whether LINK message has been received from node *j*, to which a token message was recently sent. Any height information received at node *i* from a node *j* for which *receivedLINK*[*j*] is false will not be recorded in *htVector*. Initially, $receivedLINK_i$[*j*] = true for all $j \in N_i$.
- *forming*[*j*]: a boolean array set to true when link to node *j* has been detected as just forming and reset to false when first LINK message arrives from node *j*. Initially, $forming_i$[*j*]=false for all $j \in N_i$.
- *formHeight*[*j*]: an array of triplets storing value of *i*'s height when new link to *j* first detected. Initially, $formHeight_i$[*j*]=$height_i$ for all $j \in N_i$.

The following are the messages used in the algorithm. Note that each message is attached with the *height* value, denoted by *ht*, of the node sending the message. Also note that we use "the front node of *Q*" to indicate "the node whose request message is at the front of queue *Q*."

- TOKEN(*t*): a unique message for nodes to enter the *CS*. The data field *t*, $0 \le t \le k$, of the message indicates the number of available resources.
- REQUEST(*i*, *R*): When *i* wishes to enter the *CS* to access *h* resources with priority *R*, it sends out REQUEST(*i*, *R*) to the neighbor indicated by *next*.
- RELEASE(*r*): When *i* leaves the *CS* to release *r* copies of resources, it first calls *aging* procedure to increase the priority of every request message in *Q*. And if node *i* is the token holder, it just increases *t* of TOKEN(*t*) by *r* and sends TOKEN(*t*) to the front node (if exists) of *Q*. If *i* is not the token holder, it just sends RELEASE(*r*) to the neighbor indicated by *next*.
- UPDATE(*i*, *S*): When *i* receives a new request with priority *S*, which is higher than those of messages in *Q*, it sends out UPDATE(*i*, *S*) to the neighbor indicated by *next* to update its request priority to be *S* to reflect the priority change.
- LINK: a message used for nodes to exchange their height values with neighbors.

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, or a signal from the application layer for accessing or releasing resources. Each event triggers a procedure which is assumed to be executed atomically. Below, we present the overview of the event-driven procedures:

- Requesting *h* copies of resources with priority *R*: When node *i* requests to enter the *CS* with priority *R* to access *h* resources, 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 (the single element must be REQUEST(*i*, *R*)), it calls *forwardRequest*() to send a REQUEST(*i*, *R*) message to the neighbor indicated by *next*. If node *i* holds TOKEN(*t*), *i* then checks if $t \ge h$. If so, *i* sets $t = t - h$, removes *i* from *Q* and sets *state* to *CS* to access *h* resources, since *i* will be at the front of *Q*. On the contrary, if $t < h$, then node *i* persists in waiting for the condition $t \ge h$ to be true to enter the *CS*. Note that after node *i* enters the *CS*, if *Q* is not empty, then *i* sends TOKEN(*t*) to the requesting neighbor at the front of *Q* (by calling *giveTokenToFrontOfQ*() procedure) to allow the concurrent access of resources.
- Receiving a priority update message: When a UPDATE(*j*, *S*) message sent by a neighbor *j* is received at node *i*, *i* changes the priority of *j*'s request message and adjust its position in *Q* according to the new priority if *j*'s request message is in *Q*. If *j*'s request is moved to the front of *Q* due to the priority update and *i* does not hold the TOKEN, then *i* should also send out a UPDATE(*i*, *S*) message to the neighbor indicated by *next* to report the priority change on behalf of *j*.
- Releasing *r* copies of resources: When node *i* leaves the *CS* to release *r* copies of resources, it sets *state*=*NCS*. If node *i* does not hold the token, it calls *forwardRelease*(*r*) to send out RELEASE(*r*) message to the neighbor indicated by *next*. On the other hand, if *i* holds the token TOKEN(*t*), *i* sets $t = t + r$.
- Receiving a request message: When a REQUEST(*j*, *S*) message sent by a neighbor *j* is received at node *i*, *i* ignores the message if *receivedLINK*[*j*] is false. Otherwise,

i changes *htVector*[*j*] according to the height value attached with REQUEST(*j*, *S*). And *i* enqueues the request on *Q* if the link between *i* and *j* is incoming at *i*. If *Q* is non-empty, and *state* ≠ *CS*, *i* calls *giveTokenToFrontOfQ*() provided *i* holds the token. Non-token holding node *i* calls *forwardRequest*() if $|Q|=1$ or if *Q* is non-empty and the link to *next* has reversed. If $|Q|\geq 2$ and REQUEST(*j*, *S*) is at the front of *Q*, then *i* sends out a UPDATE(*i*, *S*) message to report that the highest priority of the messages in *Q* of node *i* is changed to be *S*.

- Receiving a release message: Suppose node *i* holds the token, then when a RELEASE(*r*) message sent by a neighboring node *j* is received at node *i*, *i* sets $t=t+r$. Note that if *state* = *ES* at this time point, it means that *i* is waiting for $t\geq h$ (within the *giveTokenToFrontOfQ*() procedure) to enter the *CS*, where *h* is the number of resources *i* requests. After $t=t+1$ is executed, if $t\geq h$, then node *i* can stop the waiting and can enter the *CS*. Otherwise, node *i* keeps waiting within the *giveTokenToFrontOfQ*() procedure for the condition $t\geq h$ to be true to enter the *CS*. For the condition that node *i* does not hold the token, *i* just calls *forwardRelease*(*r*) to forward the release message when it receives a RELEASE(*r*) message.
- Receiving the token message: When node *i* receives a TOKEN(*t*) message from some neighbor *j*, *i* sets *tokenHolder* to true. Then *i* lowers its height to be lower than that of the last token holder (i.e., node *j*), and informs all its neighbors of its new height by sending LINK messages, and calls *giveTokenToFrontOfQ*() if $|Q|>0$.
- Receiving a link information message: When a link information message LINK from node *j* is received at node *i*, *j* is added to *N* and *j*'s height is recorded in *htVector*[*j*]. If *j*'s request message is in *Q* and *j* is an outgoing link, then *j*'s request message is removed 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 (reroute the 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* and sets *receivedLINK*[j] to ture. And if *j*'s request message is in *Q*, the request is deleted 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 (reroute the request) for the token.
- Link forming: When node *i* detects a new link to node *j*, *i* sends a LINK message to *j*.

Below, we introduce the procedures called by the event handling procedures mentioned above.

- Procedure *giveTokenToFrontOfQ*(): Node *i* dequeues the first element on *Q* and sets *next* equal to the first element. If *next* = *i*, then *i* checks if $t\geq h$, where *t* is the field of TOKEN(*t*) message recording the number of unoccupied resources and *h* denotes the number of resources node *i* requests. If so, *i* sets $t=t-h$ and then *i* enters the *CS*. Otherwise, *i* waits for the condition $t\geq h$ to be true. After *i* enters the *CS*, if *Q* is not empty then *i* recursively calls *giveTokenToFrontOfQ*() procedure to pass TOKEN message to the node at the front of *Q* to allow concurrent access of the resources. Now, consider the case of *next* ≠ *i*. In this case, *i* lowers *htVector*[*next*] to ($height.h_1$, $height.h_2 -1$, *next*), so that any incoming REQUEST message will be sent to *next*. Node *i* also sets *tokenHolder* to false, and then sends a TOKEN(*t*) message to *next*. If *Q* is non-empty after sending the token message to *next*, a request message REQUEST(*i*, *R*) (*R* is the priority of the request message at the front of *Q*) is sent to *next* immediately following the token message so that 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 [7] and informs all its neighbors of its height change with LINK messages. Every node *v* is deleted from *Q* if *v* is at a outgoing link. If *Q* is not empty at this point, *forwardRequest*() is called since *i* must send another request (reroute request) for the token.
- Procedure *forwardRequest*(): Selects node *i*'s lowest-height neighbor to be *next*. Sends a request message REQUEST to *next*.
- Procedure *forwardRelease*(*r*): A non-token holding node *i* selects its lowest-height neighbor to be *next* and sends a release message RELEASE(*r*) to *next*. Note that the *forwardRelease*(*r*) procedure is never called by a token-holding node.

4. Correctness

In this section, we prove that the proposed algorithm satisfies the mutual exclusion property, the highest priority first serve property, and the bounded delay property. We first show that the mutual exclusion property is guaranteed.

Theorem 1. The algorithm ensures the mutual exclusion property.
Proof:
The algorithm assumes a unique token with the field *t* for recording the number of unoccupied resources out of *k* shared resources, where $t=k$ initially. When a node wishes to enter the *CS*, it must first own the token and then checks if *t* is larger than the number of requested resources. If so, the node decreases the number of requested resources from *t* and enters the *CS*. Thus, no more than *k* resources can be accessed concurrently. The theorem holds. ■

Below, we show that the proposed algorithm satisfies the highest priority first serve (HPFS) property in Theorem 2.

Theorem 2. The algorithm ensures the highest priority first serve (HPFS) property.
Proof:

When a node receives a request, it checks whether or not the request's priority exceeds the priority of the request at the front of its local queue. If so, the priority of the received message exceeds all the priorities of the requests in the local queue. An UPDATE message is sent to *next* to report the higher priority newly found. The UPDATE message propagates along the path indicated by *next* until the token holder is reached or until the priority of the received request does not exceed the priority of the request at the front of the local queue. Nodes receiving UPDATE messages will adjust requests' positions in local queues according to the updated priorities. Thus, the token will first be passed to the node with the highest priority. According to the proposed algorithm, the node with the highest priority will hold the token until it acquires enough resources and enters the *CS*. The theorem holds. ■

Below, we prove that the proposed algorithm satisfies the bounded delay property by first showing that a requesting node owns the token eventually. Consider the logical graph whose arcs are indicted by *next* variables (from the node of a larger *height* value to the node of a smaller *height* value). Since the *next* variable stores the neighboring node with the smallest *height* value and all the *height* values are totally ordered, the logical graph has no cycles and thus 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 3 in [27].

Lemma 1. If link changes cease, the logical graph whose arcs are indicated by *next* variables is a token oriented DAG. ■

On the basis of Lemma 1, we can prove that a requesting node (a node in the *ES*) owns token eventually.

Theorem 3. The algorithm ensures the bounded delay property.

Proof:

When a token holder *i* is not in the *ES*, it passes the token to the node *j* at the front of the queue *Q*. Node *i* then removes *j* from *Q* after passing the token. Afterwards, if *Q* is not empty, *i* will send a request message to *j* so that the token will eventually be returned to *i*. Furthermore, the proposed algorithm applies *aging* procedure to increase the priorities of pending requests in queue. Thus, every node's request will eventually be of the highest priority and be at the front of the queue to have the opportunity to own the token. Since the algorithm make a node send request message to the node indicated by *next*, we have, by Lemma 1, that there is a request chain toward the token holder for every requesting node with pending request. Hence, a requesting node owns the token eventually.

According to the proposed algorithm, the node with the highest priority will hold the token and enter the *CS* when $t \geq h$, where t is the field in the token message recording the number of unoccupied resources out of totally k shared resources, and h is the number of resources the node requests, $0 \leq t \leq k$, $1 \leq h \leq k$. Since each node sends release message to the token-holding node along the path indicated by *next* pointer to add the number of released resources to t when it leaves the *CS*, the condition $t \geq h$ eventually holds and the node with the highest priority will eventually enter the *CS*.

To sum up, every node will eventually become the node with the highest priority and will eventually enter the *CS*. The theorem holds. ■

5. Concluding Remarks

In this paper, we have proposed a distributed prioritized h-out of-k mutual exclusion algorithm for a MANET with real-time or prioritized applications. The proposed algorithm can be regarded as a non-prioritized one if we assume that each node has the same priority. The MANET has the characteristic of dynamically changing topology since wireless link "failures" and/or "formation" occurs frequently. The proposed algorithm is sensitive to link forming and link breaking and thus is suitable for a MANET. However, the proposed algorithm can also be applied to distributed systems consisting of stationary nodes that communicate with each other by exchanging messages over wired links. In such a case, the assumptions 3 and 4 adopted by the RL algorithm [27] (please refer to section 2) can be omitted. The topology of the distributed system is fixed (and thus *N*, the set of all neighboring nodes is fixed), and all execution steps, messages, and variables concerning only with the link forming and breaking may be omitted (or they can be retained and remain intact when the proposed algorithm is executed).

Based on the statements mentioned above, we can draw the following conclusion: The proposed algorithm is very flexible since it can be used to solve many types of problems for many types of environments. To be more precise, the proposed algorithm can be used to solve the prioritized (or non-prioritized) h-out of-k mutual exclusion, k-mutual exclusion, and mutual exclusion problems for mobile ad hoc networks (MANETs) and distributed systems.

References

[1] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "A bounded first-in, first-enabled solution to the *l*-exclusion problem," in *Proc. 1990 Workshop on Distributed Algorithms*, pp. 422-431.

[2] Ye-In Chang, "A Priority-Based Approach to Mutual Exclusion for Read-Time Distributed Systems," in *Proc. of the 1992 International Computer Symposium*, pp. 36-43, 1992.

[3] Ye-In Chang, "Design of Mutual Exclusion Algorithms for Real-Time Distributed Systems," *Journal of Information Science and Engineering*, 10(4):527-548, Dec. 1994.

[4] X. Defago, "Distributed computing on the move:

From mobile computing to cooperative robotics and nanorobotics (a position paper)," in *Proc. of the ACM Workshop on Principles of Mobile Computing (POMC'01)*, pp. 49-55, August 2001.

[5] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, 8(9):569, Sept. 1965.

[6] M. Esbjornsson and M. Ostergren, "Hocman: Supporting Mobile Group Collaboration," in *Proc. of CHI'02*, 2002.

[7] 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.

[8] A. Goscinski, "A synchronization algorithm for processes with dynamic priorities in computer networks with node failures," *Information processing Letters*, 32:129-136, 1989.

[9] A. Goscinski, "Two algorithms for mutual exclusion in real-time distributed computer systems," *The Journal of Parallel and Distributed Computing*, 9(77-82), 1990.

[10] L. E. Holmquist, J. Falk, and J. Wigstrom, "Supporting Group Collaboration with Inter-Personal Awareness Devices," *Journal of Personal Technologies, Special Issue on Handheld CSCW*, 3(1):13-21, 1999.

[11] A. Housni, M. Tréhel, "Improvement of the distributed algorithms of mutual exclusion by introducing the priority," in *Proc. of International Conference on Information Society in the 21 Century: Emerging Technologies and New Challenges*, 2000.

[12] A. Housni, M.Tréhel, "A New Distributed Mutual Exclusion Algorithm for two Groups," in *Proc. of 2001 ACM Symposium on Applied Computing (SAC2001)*, Track on Parallel and Distributed Processing, 2001.

[13] A. Housni, M. Tréhel, "Distributed mutual exclusion by groups based on token and permission," in *Proc. of ACS/IEEE International Conference on Computer Systems and Applications*, 2001.

[14] J.-R. Jiang, "A group mutual exclusion algorithm for ad hoc mobile networks," in *Proc. of the 6th International Conference on Computer Science and Informatics*, pp. 266-270, March 2002.

[15] J.-R. Jiang, "A Distributed *h*-out of-*k* Mutual Exclusion Algorithm for Ad Hoc Mobile Networks," in *Proc. of the 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, 2002.

[16] J.-R. Jiang, "Distributed *h*-out of-*k* mutual exclusion using *k*-coteries," in *Proc. of the 3rd International Conference on Parallel and Distributed Computing, Application and Technologies (PDCAT'02)*, pp. 218-226, 2002.

[17] Y. Manabe, R. Baldoni, M. Raynal, S. Aoyagi, "*k*-Arbiter: a safe and general scheme for *h*-out of-*k* mutual exclusion," *Theoretical Computer Science.*, 193(1-2): 97-112, 1998.

[18] J. Macker and S. Corson, "Mobile Ad Hoc Networks (MANET)," *IETF WG Charter*, http://www.ietf.org/html.charters/manet-charter.html, 1997.

[19] C. Marshall, M. N. Price, G. Golovchinsky, B. N. Schilit, "Collaborating over Portable Reading Appliances," *Personal Technologies*, 3(1), 1999.

[20] Y. Manabe, N. Tajima, "(*h*-*k*)-arbiter for *h*-out of-*k* mutual exclusion problem," in *Proc. of 1999 IEEE International Conference on Distributed Computing Systems*, pp. 216-223, 1999.

[21] F. Mueller, "Prioritized token-based mutual exclusion for distributed systems," in *Proc. of 12th IPPS/SPDP Conference*, 1998.

[22] B. M. K. Qazzaz, "A new prioritized mutual exclusion algorithm for distributed systems," *Doctoral Thesis, Dept of Comp. SCI., Southern Illinois University*, Carbondale, 1994.

[23] M. Raynal, "A distributed solution for the *k*-out of-*m* resources allocation problem," *Lecture Notes in Computer Sciences*, Springer Verlag, 497:599-609, 1991.

[24] J. Redstrom, L.E. Holmquist, P. Dahlberg, and P. Ljungstrand, "Ad Hoc Information Spaces," *Technical Report*, Viktoria Institute, Sweden, 1999.

[25] A. Schmidt, H.-W. Gellersen, M. Beigl, "Handheld CSCW," in *Proc. of the Workshop on Handheld CSCW, ACMs 1998 Conference on Computer Supported Cooperative Work (CSCW)*, 1998.

[26] M. Tréhel, A. Housni, "The Prioritized and Distributed Synchronization in the Structured Groups," *ICCS*, 2001, LNCS No. 2073, pp. 294-303, 2001.

[27] J. Walter, J. Welch, and N. Vaidya, "A Mutual Exclusion Algorithm for Mobile ad hoc Networks," *Wireless Networks*, 7(585-600), 2001.