

A Distributed Mutual Exclusion Algorithm for Mobile Ad-Hoc Networks

Roberto BALDONI, Antonino VIRGILLITO, Roberto PETRASSI

Dipartimento di Informatica e Sistemistica

Università di Roma “La Sapienza”

Via Salaria 113, 00198 Roma, Italia.

{baldoni,virgi,petrassi}@dis.uniroma1.it

Abstract

A distributed mutual exclusion algorithm based on token exchange and well suited for mobile ad-hoc networks is presented along with a simulation study. The algorithm is based onto a dynamic logical ring and combines the best from the two families of token based algorithms (i.e., token-asking and circulating token) in order to get a number of messages exchanged per CS access (the main performance index for such algorithms) that tends to optimal values under heavy request load (i.e., two application messages each CS access). We present a simulation study that (i) confirms this optimality and (ii) shows that, in a mobile ad-hoc network, an effective reduction in the number of hops per application message can be achieved by using a specific policy to build on-the-fly the logical ring.

Key Words: token-based algorithms, distributed synchronization, distributed mutual exclusion.

1. Introduction

A mobile ad-hoc wireless network (MANET) is a collection of mobile nodes that communicate over paths composed of one or a sequence of wireless links. A wireless link is established only if two nodes are within a certain transmission radius. Moreover, the nodes mobility pattern creates unpredictable wireless links formation and removal, as a consequence a path between two nodes can change very frequently due to topology change.

In [1], Badrinath et al. show that also in MANET, implementing distributed algorithms that ensures control and ordering (such as mutual exclusion) is a critical point for many specific problems (e.g. channel allocation). Since these networks presents limited device power supply, the resource consumption required for the execution of distributed algorithms should be carefully controlled [2]. However, in such setting previous specific performance metrics (e.g. power

consumption) as well as communication issues (e.g. link failures and recovery) have to be taken into account.

One of the most classical paradigms of distributed computing is Distributed Mutual exclusion (DMUTEX). It consists in defining a protocol run by a set of processes which want to coordinate themselves to access a single shared resource (or a piece of code), namely *critical section* (CS), that can be used only by one process at a time. In the recent years, several DMUTEX algorithms specifically designed for MANETs have already been presented [1, 13, 12]. These algorithms can adapt their behaviour to changes in the physical topology of the network to reduce communication overhead in a mobile environment.

In this paper we present a general DMUTEX algorithm presenting features that allow it to be effectively used over a MANET. The algorithm aims at maintaining device power consumption as low as possible by reducing the number of hops traversed per CS execution and by not sending any control message when no processes request the CS. Mobility is addressed by exploiting the information of the routing table in order to send each message to the closest node in terms of number of hops.

Our DMUTEX algorithm can be classified as “token-based” (e.g [5, 6, 7, 8, 10, 11]): Token-based algorithms allow the process holding the token to enter its CS¹. These algorithms can be further split into two families: “Token-Asking” and “logical ring” [9]:

- **Token-asking** (e.g. [5, 7, 8, 10, 11]): A requesting process broadcasts a request for the token to all the processes. The process owning the token inserts the request in a request queue and when it leaves the CS it sends the token to the first process in the queue, together with the request queue. When the request queue is empty, the process stops the token and waits for a request. In such a case we say that the system enters in an idle state. In other words idle

¹In “non-token-based” algorithms (also defined “Permission Based”) each process has to get an explicit grant, by means of a message, from a set of other processes before entering its CS. References to some of such algorithms can be found in [9].

states mean no requests and no messages exchanged by the DMUTEX algorithm.

- **Logical ring** (e.g. [6]): The token circulates perpetually around a logical ring using point-to-point messages. A process receives the token from its predecessor in the ring, accesses the CS, if needed, and then sends the token to its successor on the ring. In the best scenario, CS access can be achieved with very low overhead in terms of messages exchanged when all processes issue a request in the same round of the token, i.e. one message for CS entry.

The algorithm presented in this paper belongs to the token-asking family. However, it structures processes onto a logical ring by allowing CS access by means of a circulating token (as logical ring algorithms). This ensures that circulating-token optimal scenarios could happen.

The logical ring of processes is dynamic: each time a process receives the token from its predecessor, it decides on-the-fly which will be its successor between processes that did not yet receive the token during the token round. This decision is based on a given policy \mathcal{P} . This policy can be specialized to adapt the algorithm to specific settings. In this paper we chose \mathcal{P} to order processes according to their hop distance. Each round of the ring has a coordinator which is the only process that can block the token and enter an idle state. The aim of a round of the token is to inform processes about coordinator changes. Experimental performance results, obtained by simulation, shows that the proposed DMUTEX algorithm running on a mobile network achieves best performance when request load is heavy.

The rest of the paper is structured into four sections. Section 2 describes the system model. Section 3 explains the proposed DMUTEX algorithm. Section 4 presents the results of the simulation study. Section 5 concludes the paper.

2. System model

We consider a distributed system composed by a set of $n \geq 1$ processes $\{p_1, \dots, p_n\}$ running each on a different mobile node that follows an unknown mobility pattern. Processes cooperate only through message exchanging and do not have access to a shared memory. The DMUTEX algorithm is formed by the concurrent execution of n identical protocols one at each process. Processes do not crash and network is reliable (i.e., messages are eventually delivered by the network to the intended receiver). There is no assumption about the message transfer delays or the speed of execution of a process, so this makes the distributed system *asynchronous*. The network is not subject to permanent partitions, that is even if a partition occurs each pair of processes will be eventually able to communicate. Finally, we assume that a process can query at each time the information provided by the routing protocol.

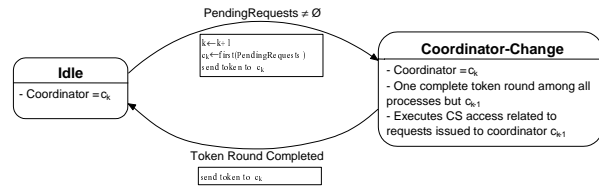


Figure 1. Algorithm state diagram

3. The DMUTEX Algorithm

The algorithm, from an external viewpoint, continuously executes transitions between two states: *Idle* and *Coordinator-Change*. The algorithm evolves in a sequence of rounds during which the two states alternate. A new round starts when the state switches to *Coordinator-Change*.

For each round of the algorithm, one process is declared the *coordinator*. We will indicate with c_k the coordinator for round k . The coordinator changes from c_{k-1} to c_k each time there is a transition between *Idle* and *Coordinator-Change* state and the process p_i that provokes the transition becomes the current coordinator c_k . The latter is the only process enabled to execute the next transition from *Coordinator-Change* to *Idle*. The state diagram is described in Figure 1.

a. Algorithm’s State Diagram. A process that wants to access its critical section, issues a request to the coordinator and waits for the token. The coordinator inserts the request in a set, namely $pendingRequest(\mathcal{P})$, ordered according to a certain serialization discipline \mathcal{P} . The serialization discipline \mathcal{P} could be deterministic (e.g. the process in the set with the lowest identifier) or driven by some specific knowledge (e.g. FIFO, short-job-first, the closest process to p_i in terms of some metric such as the number of hops etc.). Initially the algorithm is in the *Idle* state and the coordinator c_0 corresponds to the process p_1 and this information is known by all the processes.

Transitions from Idle state to Coordinator-Change state: it occurs when the coordinator c_k is in the *Idle* state and the set $pendingRequest(\mathcal{P})$ is not empty. During the transition, the coordinator sends the token to the first process in the ordered set $pendingRequest(\mathcal{P})$, say p_j , that will become the coordinator c_{k+1} .

Coordinator-Change state. This state consists of one round of the token on a logical ring. The ring starts from the current coordinator c_k and it is composed of $n - 1$ processes (the process corresponding to the previous coordinator c_{k-1} is excluded from the ring). This round has two targets: (i)

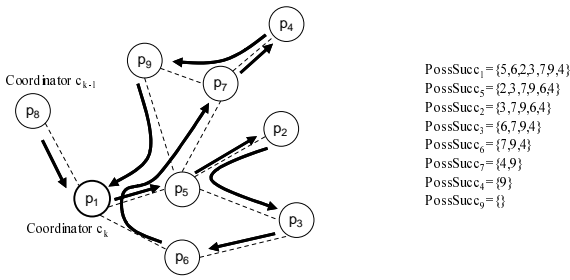


Figure 2. Example of the logical ring

informing all the processes about the identifier of the current coordinator c_k and (ii) allowing requesting processes to access their CS once they receive the token. These requests were issued to the previous coordinator c_{k-1} .

Transition from Coordinator-Change state to Idle state. It occurs when the current coordinator c_k receives back the token from its predecessor in the logical ring embedded in the Coordinator-Change state.

b. Logical Ring The main task of the Coordinator-Change state is to execute a round of a logical ring of $n - 1$ processes. The round starts from the coordinator c_k and does not include c_{k-1} . As a consequence the structure of the logical ring has to be computed on-the-fly. i.e., the process p_i that receives the token has to relay it towards a process (p_i 's successor) that (i) has not received the token yet in this round and that (ii) is distinct from c_{k-1} . The last process in this logical ring relays the token to c_k provoking the transition of the algorithm to the Idle state.

As a consequence the structure of the ring is dynamic and might be different in each round, adapting itself to the mobility of nodes. Operationally the processes passed by the token in a round of a logical ring are kept in a data structure, actually a vector of Boolean of size n , sent with the token, namely *receivedToken*. *receivedToken*[i] is FALSE if p_i has not yet received the token in the current round. During the transition from the Idle state to the Coordinator-Change state, the coordinator c_{k-1} , say process p_i , sets each entry of *receivedToken* to FALSE except *receivedToken*[i], which is set to TRUE before sending out both the token and the data structure *receivedToken* to the new coordinator c_k . *receivedToken*[i] equal to TRUE avoids that the token during the round in the Coordinator-Change state will pass through p_i .

When a process p_i receives the token along with the data structure *receivedToken* during the round coordinated by c_k , it first sets *receivedToken*[i] to TRUE, then defines a set of possible successors $possSucc_i$ in the following way:

$$possSucc_i = \{p_\ell | receivedToken[\ell] = FALSE \wedge (\ell \neq i)\}$$

If $possSucc_i$ is empty then the token is sent to the coordinator c_k (i.e., the round is finished and the algorithm is going to enter Idle state). Otherwise the set is ordered according to a certain serialization discipline \mathcal{P}^2 which provides a total order on $possSucc_i$, denoted $possSucc_i(\mathcal{P})$, based on the distance in terms of number of hops between each process in the set and p_i . This distance is calculated by p_i basing on information provided by the underlying routing protocol. If a node is temporarily disconnected, it is considered to be at infinite distance. If two processes are at the same distance, the one with the lowest identifier comes first. The first process in the ordered set $possSucc_i(\mathcal{P})$, say p_j , is selected and the token is sent to it. If $possSucc_i(\mathcal{P})$ contains only processes at infinite distance, p_i waits and tries to retransmit after a certain amount of time. For the assumptions made in Section 2, p_j will be eventually reachable and will receive the token.

In this way, the ring is built dynamically by always passing the token to the node that requires the lowest effort in terms of network resources and power supply. Of course this discipline does not ensure optimal behaviour. However, it provides a good heuristic, actually a simple greedy policy, that takes into account mobility with the only additional assumption of querying the local routing table. If a more sophisticated behaviour is required, \mathcal{P} can be easily substituted without impacting the rest of the algorithm.

Figure 2 shows an example of how the algorithm builds a logical ring. In the example, each process in the system is connected to any other. Processes in $possSucc_i(\mathcal{P})$ will be ordered according to the hop count between p_i and all the other processes, calculated by the routing layer at the time the token have to be sent. Then, when p_i sends the token it chooses as receiver the closest process among the ones that have not received the token yet. The Figure depicts the physical configuration of processes. A link (thin dotted line) between two processes indicate that they are in each other's transmission range. In this example, we assume for clarity that nodes do not move. The thick arrows highlight the actual logical ring formed by the sequence of token transmissions.

c. Local Protocol run at a process The explanation of the protocol is split into three parts: first, the data structures maintained at each process are explained, second, the list of messages exchanged is given and, third, the process behavior is presented.

Data Structures. Each process p_i in the system maintains the following set of local data structures³:

- *coordinator*: identifier of the current coordinator process

²In the rest of the paper, for simplicity, we assume that this serialization discipline is the same used to order the set *pendingRequests*.

³For simplicity sake we omit using the subscripts when identifying local data structures at a generic process p_i .

to the best knowledge of p_i . Initially it is set to p_1 at each process.

- *state*: is a flag which value can be IDLE or NOT-IDLE. *state* is set to IDLE only if p_i is the coordinator and the system is in the Idle state (i.e. the token has been stopped), otherwise it is set to NOT-IDLE.

- *tokenHolder*: is a boolean variable which is set to TRUE if the process receives the token and can enter the CS. Initially it is set to TRUE in p_1 and to FALSE in the other processes.

- *requestCS*: is a boolean variable which is set to TRUE if the process has a pending request for accessing the CS. Initially it is set to FALSE.

Moreover p_i also maintains a boolean vector, *receivedToken*[], initialized to FALSE at each entry, a queue *pendingRequest*(\mathcal{P}), initialized to empty, and *possSucc*(\mathcal{P}), a set initialized to empty, whose meaning has been explained in the previous sections. Finally, the function *successor*() used by process p_i to define on-the-fly its successor in the logical ring is defined by the following pseudo-code:

```
Function successor()
  compute possSucc( $\mathcal{P}$ );
  if possSucc( $\mathcal{P}$ ) =  $\emptyset$ 
    then return coordinator;
  else return first(possSucc( $\mathcal{P}$ ));
```

d. Messages. A process p_i communicates with other processes by sending one of the following messages:

- *Token*(*type*, *rt*, *coordinator*) whose meaning is to pass the token to a selected process. The message contains three fields: *rt* which stores the value of *receivedToken* vector of the sender, *coordinator* which stores the value of the *coordinator* variable of the sender and the *type* field, that can assume two values: COORDINATOR-CHANGE or NEW-COORDINATOR. If *type* = NEW-COORDINATOR, it identifies the message that executes the transition from Idle state to Coordinator-Change state of the DMUTEX algorithm. If *type* = COORDINATOR-CHANGE, it indicates that the DMUTEX algorithm is in the Coordinator-Change state.

- *Request*: p_i issues a MUTEX request to the coordinator.

If, when sending one of such messages, the destination process is unreachable, the sending process will wait for a finite amount of time and then try to transmit the message again, until it succeeds.

e. Behaviour of a process. The protocol run at each process is composed of a procedure to request the shared resource (Figure 3) and two message handlers (Figure 4) to treat the receipts of token and request messages respectively.

Procedure *requestingCS*():

- (1) *requestCS* \leftarrow TRUE;
- (2) if *coordinator* \neq i then send *Request*() to *coordinator*;
- (3) Wait for *tokenHolder*;

Critical Section

- (4) *requestCS* \leftarrow FALSE;
- (5) if *state* = NOT-IDLE then
- (6) *tokenHolder* \leftarrow FALSE; % continue logical round %
- (7) send *Token*(COORDINATOR-CHANGE, *receivedToken*, *coordinator*) to *successor*();
- (8) endif

Figure 3. Code of *requestingCS* procedure

In the following we describe procedures associated to each of such events.

- *requestingCS*() procedure (Figure 3): When a process wants to access to its CS, it sets *requestCS* to true and sends a *Request* message to the coordinator. If p_i is the coordinator, it skips this sending (Line 2). Then p_i waits for the token and enters its CS. Upon exiting from the CS it sets *requestCS* to false and if the algorithm is in the Coordinator-Change state (i.e., not in the Idle state), p_i forwards the token to its successor in the logical ring. Note that if p_i has *state* set to TRUE, this implies that p_i is the coordinator and its *pendingRequest*(\mathcal{P}) is empty. In such a case, if p_i issues a request it can access the CS without notifying it out. Finally lines 1-2, 4-8 are executed atomically.

- *Request message handler* (Figure 4): When a process p_i receives a request message from a process p_j , if it is the current coordinator, it inserts the request in the set *pendingRequests*(\mathcal{P}) (line 9). Otherwise the request is discarded.

- *Token message handler* (Figure 4): When a process i receives a token message, first it executes (line 10) the updates of local variables that does not depend from the field *type*. Then three alternatives are possible and, for simplicity, they are handled by a case structure:

1. The process receives the token at the end of a logical round. It enters the Idle state and remains there until the set *pendingRequests*(\mathcal{P}) is empty and p_i is not accessing its CS. Once a request arrives at p_i (line 9), the algorithm executes the transitions from the Idle state to the Coordinator-Change state (lines 15-18). More specifically, the new coordinator is identified as the first process in the ordered set *pendingRequests*(\mathcal{P}), the data structure that manages the round of the logical ring within the Coordinator-Change state is initialized and the token is sent to the process identified by the *successor*() function setting NEW-COORDINATOR as *type*. Note that the value of *receivedToken*[i] remaining to TRUE ensures that the token will not pass during the logical round through the old coordinator another time.

2. The process receives the token with the field *type*

```

When Request() is received from  $p_j$ :
(9) if coordinator= $i$  then
    pendingRequests( $\mathcal{P}$ )  $\leftarrow$   $\{p_j\} \cup$  pendingRequests( $\mathcal{P}$ );

When Token(type, rt, c) is received from  $p_j$  :
(10) coordinator  $\leftarrow$  c; receivedToken  $\leftarrow$  rt;
    receivedToken[i]  $\leftarrow$  TRUE; tokenHolder  $\leftarrow$  TRUE;
(11) case
(12)   type = COORDINATOR-CHANGE and coordinator =  $p_i$ :
(13)     state  $\leftarrow$  IDLE; % enter Idle state %
(14)     wait ((pendingRequest  $\neq$   $\emptyset$ )  $\wedge$  ( $\neg$ requestCS))
(15)     coordinator  $\leftarrow$  first(pendingRequest( $\mathcal{P}$ ));
(16)      $\forall l \neq i$  in  $1..n$  receivedToken[l]  $\leftarrow$  FALSE;
(17)     state  $\leftarrow$  NOT-IDLE; tokenHolder  $\leftarrow$  FALSE;
(18)     send Token(NEW-COORDINATOR, receivedToken, coordinator)
        to coordinator
(19)   type = NEW-COORDINATOR: % enter Coord-Change state %
(20)     pendingRequest( $\mathcal{P}$ )  $\leftarrow$   $\emptyset$ ;
(21)   type = COORDINATOR-CHANGE  $\wedge$  coordinator  $\neq$   $p_i$ :
(22)     if  $\neg$ requestCS then
(23)       tokenHolder  $\leftarrow$  FALSE; % continue logical round %
(23)       send Token(COORDINATOR-CHANGE receivedToken, coordinator)
          to successor()
(24)     endif;
(25) endcase

```

Figure 4. Message handler code of p_i

equal to NEW-COORDINATOR. p_i detects itself to be the new coordinator, and that the algorithm moved to Coordinator-Change state. As a consequence, it empties the $pendingRequests(\mathcal{P})$ set. Note that if p_i enters this alternative, it implies that $requestCS$ is equal to TRUE (see the correctness proofs in []). Then p_i accesses the resource as soon as line 10 is executed (i.e., $tokenHolder$ is set to TRUE). The relaying of the token to the successor of p_i in the logical ring is then executed by the lines (6) and (7) belonging to the $requestingCS()$ procedure as $state$ variable in p_i is set to NOT-IDLE.

3. The process receives the token with the $type$ field equal to COORDINATOR-CHANGE and it is not the current coordinator. In this case the token can continue the logical ring within the Coordinator-Change state. In the case that p_i requested the resource (i.e., $requestCS = TRUE$) the relay of the token to the successor on the logical ring will be executed by line (7) as the variable $tokenHolder$ was set to TRUE at line (10) unblocking the $requestingCS()$ procedure from line (3). If $requestCS = FALSE$ the relay of the token is executed at line (23).

Let us finally remark that when entering one of the last two alternatives of the case structure, all lines of the message handler are executed atomically. Due to the presence of the wait statement (line 14), when entering first alternative lines 12-13 and 15-18 are executed atomically. The formal proof is omitted for brevity. It can be found in [4].

4. A Simulation Study

The simulation of the algorithm presented in this paper has been carried out using the GloMoSim [14] simulation

environment. GloMoSim is a library-based simulator for wireless and mobile network, developed in the UCLA Parallel Computing Laboratory, built upon the Parsec [3] parallel discrete-event simulation tool. We implemented our algorithm as an application-level Glomosim library. The \mathcal{P} policy is implemented by accessing the routing table of the network level protocol (Bellman-Ford) in order to choose the node at a closest hop distance when sending each message.

Simulation Settings. The simulation environment is composed by 20 mobile nodes randomly spread over an area of 700×700 meters. The transmission radius of each node is 200 meters. The dimension of the mobility area has been chosen after several trials as a tradeoff between the probability of temporary partition and the number of hops necessary for transmitting an application level message. This dimension minimize the probability of temporary partitions while maximizing the number of hops for each application message. Under such conditions it is possible to evaluate the effect of the application of the \mathcal{P} policy, that aims at minimizing the number of hops per message transmission.

Mobility of node is modeled with a random-waypoint behaviour, i.e. a process moves towards a randomly selected point inside the area and then pauses for a selected amount of time (pause time) before moving again. Nodes move at an average speed of 20 m/s inside the 700×700 meters area. The simulation has been carried out under three different mobility settings, chosen by adjusting the pause time so that the percentage of the total simulation time the node does move (calculated considering the average time required for a movement) is 100% (high mobility), 50% (medium mobility), 10% (low mobility) respectively. Processes issue requests randomly, using a Poisson distribution with mean λ which represents the number of requests/sec. generated by a single process.

Simulation Results. The results of the simulation are shown in Figures 5 and 6. Figure 5 shows the number of network-level messages (hops) per CS access as a function of the request load. This plot confirms that the algorithm performs best as request load increases. This can be explained intuitively as follows (for a formal performance analysis see [4]): under heavy request load conditions each process is continuously issuing requests, then during a token round each process is likely to be waiting for the token and can enter the critical section. So all the application messages required for the token round are effectively used to satisfy CS requests.

At very low request rates, the number of hops per CS access maintains around a approximately constant value. In this setting, for each reply from the coordinator a fixed overhead is accounted of a full token round with no pending requests served (at least $n - 2$ hops, for a system with n processes). This is confirmed by our results: in a settings

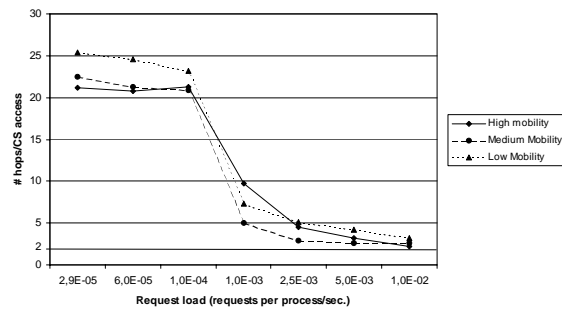


Figure 5. Hops per CS access vs. load

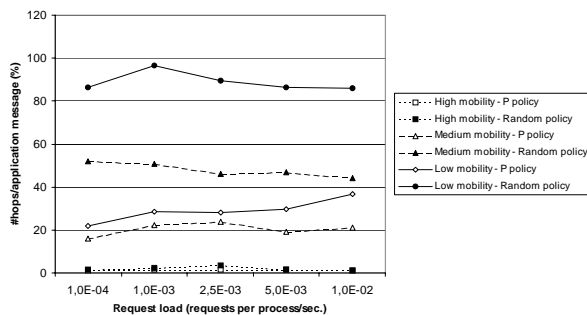


Figure 6. Hops per application message

with 20 processes, by decreasing the request load the number of hops per CS access tends to a value always higher than 20. From the plot is evident that mobility has no significant impact on this measure.

Figure 6 plots the average number of hops, in percentage, per application-level messages sent, vs the request load. Six different series of values are obtained: in each of the three mobility settings defined above, the logical ring is built with 2 different policies, \mathcal{P} (the closest node is chosen as next in the ring) and a simple random policy (a random node is chosen as next in the ring). The plot shows that the difference in percentage between hops and application messages increases when node mobility is lower. This means that under high mobility any two nodes are within one hop with very high probability and that the information provided by the routing table is always stale. This makes \mathcal{P} and the random policy in this case equivalent.

However, as mobility decreases it is clear that \mathcal{P} provides an effective performance improvement with respect to the random policy, reaching a reduction of 60% under low mobility conditions, as the information provided by the routing table allows to select with high probability the actual closest node. These measures result independent from the request load, because the number of hops increases proportionally with the number of application messages sent.

5. Conclusions

A simple distributed mutual exclusion algorithm based on token exchange well suited to mobile ad-hoc network has been presented. The algorithm combines the best from the two families of token based algorithms (i.e., token-asking and circulating token) getting under heavy request load a number of hops traversed per CS very close to an optimal value. Moreover it is based on a dynamic logical ring, built on-the-fly built by using a policy \mathcal{P} at each process that selects as the next process in the ring the closest one in terms of numbers of hops. In this paper we showed that in mobile ad-hoc networks \mathcal{P} reduces as many as possible the number of hops experienced per CS execution in a algorithm round.

References

- [1] B.R. Badrinath, A. Acharya, and T. Imielinski, *Structuring distributed algorithms for mobile hosts*, Proceedings of the The 14th International Conference on Distributed Computing, 1994, pp. 21–28.
- [2] B.R. Badrinath and T. Imielinski, *Mobile wireless computing: Challenges in data management*, Communication ACM **37** (1994), no. 10, 18–28.
- [3] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song, *Parsec: A parallel simulation environment for complex systems*, Computer **31** (1998), no. 10, 77–85.
- [4] R. Baldoni and A. Virgillito, *A token-based mutual exclusion algorithm for mobile ad-hoc networks*, Tech. Report 28-01, Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", available at <http://www.dis.uniroma1.it/~virgi>, 2001.
- [5] J.M. Helary, A. Mostefaoui, and M. Raynal, *A general scheme for token and tree based distributed mutual exclusion algorithm*, IEEE Transactions on Parallel and Distributed Systems **5** (1994), no. 11, 1185–1196.
- [6] G. Le Lann, *Distributed systems: towards a formal approach*, Proceedings of the IFIP Congress (1977), 632–646.
- [7] M. Naimi and M. Trehel, *An improvement of the log(n) distributed algorithm for mutual exclusion*, Proceedings of the 7th International Conference on Distributed Computing Systems, 1987, pp. 371–377.
- [8] K. Raymond, *A tree-based algorithm for distributed mutual exclusion*, ACM Trans. on Computer Systems **7** (1989), no. 1, 61–77.
- [9] M. Raynal, *Simple taxonomy for distributed mutual exclusion algorithms*, ACM Oper. Systems Rev **25** (1990), 189–193.
- [10] M. Singhal, *A heuristically-aided algorithm for mutual exclusion in distributed systems*, IEEE Trans. on Computers **38** (1989), 651–662.
- [11] I. Suzuki and T. Kasami, *A distributed mutual exclusion algorithm*, ACM Trans. Comput. Systems **3** (1985), 344–349.
- [12] 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), 2001.
- [13] J. Walter, J. Welch, and N. Vaidya, *A mutual exclusion algorithm for ad hoc mobile networks*, accepted to the ACM and Baltzer Wireless Networks Journal special issue on DialM papers, 2001.
- [14] X. Zeng, R. Bagrodia, and M. Gerla, *Glomosim: a library for parallel simulation of large-scale wireless networks*, Proceedings of the 12th Workshop on Parallel and Distributed Simulations (PADS '98), 1998.