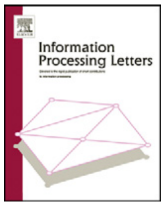
Information Processing Letters 111 (2011) 841–847

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl



A hybrid token-based distributed mutual exclusion algorithm using wraparound two-dimensional array logical topology

Hoda Taheri[a], Peyman Neamatollahi[a,*], Mahmoud Naghibzadeh[b]

[a] Department of Computer Engineering, Young Researchers Club, Mashhad Branch, Islamic Azad University, Mashhad, Iran
[b] Department of Computer Engineering, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran

ARTICLE INFO

Article history:
Received 29 October 2010
Received in revised form 25 May 2011
Accepted 30 May 2011
Available online 2 June 2011
Communicated by F.Y.L. Chin

Keywords:
Token-based algorithm
Mutual exclusion
Distributed systems
Hybrid algorithm
Concurrency
Message exchange

ABSTRACT

In token-based distributed mutual exclusion algorithms a unique object (token) is used to grant the right to enter the critical section. For the movement of the token within the computer network, two possible methods can be considered: perpetual mobility of the token and token-asking method. This paper presents a distributed token-based algorithm scheduling mutually exclusive access to a critical resource by the processes in a distributed network. This network is composed of N nodes that communicate by message exchanges. The proposed hybrid algorithm imposes a logical structure in the form of wraparound two-dimensional array on the network. It applies the concept of perpetual mobility of the token in columns and token-asking in rows of the array. The major purpose of the algorithm is to increase the scalability property and decrease overhead due to additional communication in a system with at least one unresponded critical section request at any given time. In this status, typically, the number of message exchanges is between $\sqrt{N}$ and $2\sqrt{N}$ under light demand and reduces to $\sqrt{N}$ message exchanges under heavy demand. Therefore, it outperforms lots of well known algorithms in terms of number of messages exchanged. The algorithm satisfies safety and liveness properties.

1. Introduction

A Distributed System (DS) consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. One of the most important purposes of the distributed systems is to provide an efficient and convenient environment to share resources [15]. Therefore, it is possible that more than one process request a shared resource through their critical sections simultaneously. Each process has a code segment, called Critical Section (CS), in which the process can access the shared resource. There are many situations within operating systems, distributed shared memories, distributed databases, etc., that a resource should be given to only one process at a time. When a process has to read or update certain shared data structures, it first enters a CS to achieve mutual exclusion and ensures that no other process will use the shared data structures at the same time [17]. If a resource needs to be accessed exclusively, Mutual Exclusion (ME), some controls are necessary to assure that only one process can use a shared resource at any given time. The algorithms designed to ensure ME in distributed systems are termed Distributed Mutual Exclusion (DME) algorithms. In a DS, any given node has only a partial or incomplete view of the total system [19]. So, DME problem has to be solved by using message exchanges. The problem of ME has been fairly well studied in distributed systems. The proposed solutions can be classified in token-based and non-token-based algorithms. In token-based DME algorithms, token is a unique entity in the entire system which is used to grant a node to enter

* Corresponding author.
E-mail addresses: h.taheri.mshd@gmail.com (H. Taheri), neamatollahi.peyman@gmail.com (P. Neamatollahi), naghibzadeh@um.ac.ir (M. Naghibzadeh).

0020-0190/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2011.05.021

its CS from among other nodes that are attempting to invoke their critical sections.

In this paper, a wraparound two-dimensional array logical topology is used to decrease the number of message exchanges. We attempt to propose a hybrid token-based algorithm to solve the DME problem in order to decrease communication overhead and increase scalability property in a system with at least one unresponded CS request, at any given time. This hybrid method applies the concept of perpetual mobility of the token in columns and token-asking in rows of the array. A CS entry request message is horizontally sent to the nodes in a row and the token vertically circulates in the array. The role of the common node between the row consisting of the requesting node and the column consisting of the token is to directly send the token to the requesting node. This approach satisfies two important ME necessities: safety and liveness. Also, it will be shown that the number of message exchanges becomes between $\sqrt{N}$ and $2\sqrt{N}$ under light demand; on the other hand, $\sqrt{N}$ message exchanges under heavy demand. In both cases, the algorithm acts better than many famous algorithms (e.g. [1,6,8,14,16]).

The rest of this paper is organized as follows: Section 2 provides the related works, Section 3 defines the assumptions of the algorithm, Section 4 includes the outline, data structures and messages, and algorithm behavior (the pseudo code of the algorithm is shown), Section 5 proves the correctness of the algorithm (safety and liveness properties), Section 6 calculates the performance of the algorithm for both light and heavy load conditions, Section 7 discusses about the logical topology and the application of the proposed algorithm for ad hoc networks, and a conclusion is provided, at the end.

2. Related works

Solving the ME problem (which is first introduced by Dijkstra [5]) has been one of the topics which have received the attentions of many researchers. In distributed solutions, there are two families of algorithms which are token-based and non-token-based algorithms. In token-based algorithms a simple concept is used; as only one process at a time can enter its CS (safety property), the right to enter is materialized by a special object which is unique in the whole system, namely a token. Processes requesting to enter their critical sections are allowed to do so when they possess the token. Therefore, the token gives a process the privilege of entering the CS. At any given time, the token must be possessed by one process at most. The safety property is trivially ensured as the token is unique. The only thing one has to manage is the movement of the token from one process to another so that each request can be granted eventually (liveness property). At this point, two possibilities can be considered for such a movement: the perpetual mobility of the token and the token-asking method [13].

In the perpetual mobility, the token travels from one process to another to give them the right to enter their critical sections exclusively, without paying attention to whether that process needs the token or not. Therefore, additional processing and communication are imposed on the system as overhead, especially in the light load situations in which a very few number of processes attempt to invoke their critical sections simultaneously, but the perpetual mobility of the token is very effective on the high load situations. Token-ring algorithm [7] is one of these algorithms. In this algorithm, perpetual mobility of the token on a unidirectional logical ring ensures the liveness. The main problem of this method is that it does not have the scalability property. The reason is that, by increasing the number of processes, the average waiting time for the process attempting to get the token increases.

In token-asking methods (e.g. [10,12,16]), a process which is attempting to invoke its CS, if it is not the token-holding process, requests to receive the token and waits for the token arrival. After completing the execution of its CS, the token-holding process chooses a requesting process and sends the token to it. If no process wants to use the token, the token-holding process does not need to send the token away. Using this method, Suzuki and Kasami [16] presented an algorithm that process P_i which is attempting to invoke its CS, broadcasts requested message to all other processes, $N-1$ message exchanges are required, and the token is sent directly to process P_i for which one message exchange is required. Hence, this algorithm requires N message exchanges per CS invocation at the most. We have used their scheme for the rows of the logical topology in our algorithm, with some modifications.

3. Assumptions

What we will present in this paper solves ME problem in a DS composed of N nodes and no shared memory. These nodes communicate through asynchronous message passing on a communication network layer that is error-free. At first without loss generality, we assume that only one process is in each node. Therefore, we use process and node to represent the same concept.

Message propagation delay is unpredictable but it is finite; it indicates that every message will eventually be received. This assumption prevents introducing message acknowledgement protocols. The messages are not guaranteed to be delivered in the same order as they are sent. A unique non-zero identification number is assigned to each process. Every process can send messages directly to all other processes using their identification numbers, which are considered as their addresses (complete communication graph similar to [2,6,10,11,14,16]).

The *token* mobility (which is the key feature of our algorithm) is based on a two-dimensional logical topology. More precisely, every node belongs to two logical rings. In the first one, the *token* visits the nodes sequentially in vertical manner, while in the second one nodes are visited on demand in horizontal manner. Besides, each request issued by a node must be broadcast to all other nodes located on the second ring. Therefore, the logical structure of the interconnecting network is a wraparound two-dimensional array: the token moves perpetually from one node to another along the vertical rings (circles) and it moves on demand along the horizontal rings (circles). The algorithm does not entail any specific physical interconnection topology. It is assumed that $N = d^2$, where d is an integer and N

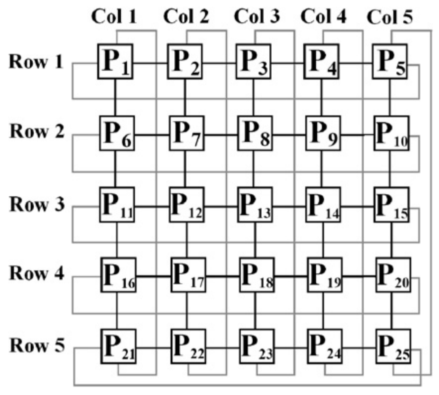


Fig. 1. A proposed network of 25 nodes.

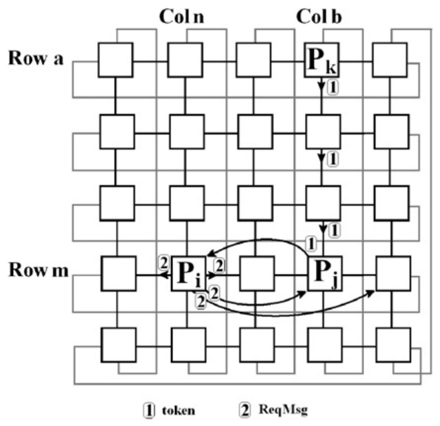


Fig. 2. Total messages exchanged between processes in the algorithm.

is the number of processes. Therefore, the logical interconnecting array is composed of $\sqrt{N}$ rows and $\sqrt{N}$ columns. Every process knows its row and column numbers in the wraparound two-dimensional array. In the sample system shown in Fig. 1, every node knows other nodes in its row and also its down neighbor.

We assumed that processes operate correctly. A process has the permission of dedicated access to the resource only when executing its CS but for a limited amount of time. While a process requests its CS, it cannot create another request for the CS until the first one is granted. We assumed that at least one unresponded request exists in proposed DS, at any given time, and also CS entry requests might not be satisfied in the order of their construction, like algorithms proposed in [1,2,7,8,10,12,16].

4. The new algorithm

The explanation of the algorithm is divided into three parts: first, the outline is described, second, data structures and messages are introduced, and third, the overall algorithm is presented.

4.1. The outline

By hybrid we mean the concept of perpetual mobility of the *token* in columns and token-asking in rows of wraparound two-dimensional array. A node requests ME by informing all other nodes in its row. On the other hand, the *token* perpetually circulates from row to row. When the *token* reaches at a given row, it acquires knowledge of pending requests for that row (first stage) and serves those requests (second stage). During the second stage, the pending requests are responded. To avoid the *token* oscillate in the same row when rest of the nodes are starved, the new requests are ignored in this stage. These requests are pended until the *token* meets this row in the future.

We assume that in the beginning of the algorithm, P_k is the token-holding process (which is in Row a and Column b of the logical topology). To simplify, assume there is only one non-token-holding process, say process P_i in Row m and Column n so that m is unequal to a, which is attempting to invoke its CS. The given position of these two nodes and messages exchanged between them in the following scenario is shown in Fig. 2. Process P_i sends its CS entry request (*ReqMsg*) to all nodes in its row, then waits until it receives the *token*. All existing nodes in Row m after receiving that *ReqMsg* know that process P_i is waiting for achieving the *token*.

On the other hand, process P_k sends the *token* to the next node below. In this way, the *token* continues the perpetual mobility around Column b until it reaches one of the nodes in Row m, say process P_j, eventually. Because process P_j knows that process P_i has a pending request, after receiving the *token* forwards it to process P_i.

When process P_i receives the *token*, it enters its CS. After releasing the CS, it sends the *token* in down direction. So the *token* can continue its perpetual mobility.

4.2. Data structures and messages

ReqMsg is a message type which is sent by a process, say process P_i, to invoke its CS. This message is composed of the identification number of that process, i, and its sequence number, SN_i, which is shown by $ReqMsg(i, SN_i)$. SN_i is a counter that process P_i increases by one when attempts to invoke its CS to indicate that there is a request from this process which is unresponded.

The *token* is a record which is sent by a message. It is composed of a FIFO queue named *next* (including not yet responded requests) and an array with N elements named *seqnum*. The ith element of this array counts the number of process P_i's CS entries. The *token.seqnum* is used to distinguish responded requests from not yet responded ones, similar to the LN array in [16]. *RTR* is the other field of this record. The *token.RTR* is the number of last row which is visited by the *token*.

Each node has a queue which is composed of *ReqMsg*s named *Waiting*. It is possible that every element may be removed from everywhere in *Waiting*. Every node knows its down neighbor which is represented by constant identifier placed in *Down* variable. A local variable named *CS-permission* contains zero value or the node identifier to indicate whether a node can enter its CS or not.

For the following section, we assume that node k, k is constant, in Row a and Column b is the token-holding node, $1 \leqslant a \leqslant \sqrt{N}$ and $1 \leqslant b \leqslant \sqrt{N}$. For details, see the Initialization part in Fig. 3.

4.3. Algorithm behavior

We investigate the behavior of the algorithm in three cases (1) process P_i requests to enter its CS, (2) process P_i receives a message from process P_j, and (3) process P_i leaves its CS.

Requesting the CS: Process P_i (in Row m and Column n) which wants to execute its CS, increases SN_i by one and creates $ReqMsg(i, SN_i)$. Process P_i, after inserting $ReqMsg(i, SN_i)$ in its *Waiting* queue, $Waiting_i$, sends $ReqMsg(i, SN_i)$ to all other nodes in its row to inform them of its request. Then, process P_i waits until it receives the *token*. Whenever it receives the *token*, it sets $CS\text{-}permission_i$ to i and executes its CS. For details, see Lines 1 to 8 in Fig. 3.

Receiving a Message: When process P_i receives a message from process P_j (in Row x and Column y), two kinds of status are possible:

1. The received message is $ReqMsg(j, SN_j)$. In this case, the message is inserted in $Waiting_i$. Note that process P_j has sent its *ReqMsg* to all nodes in its row, including process P_i, to inform them of its request. For details, see Lines 19 and 20 in Fig. 3.
2. The received message is composed of the *token*. In this case, two kinds of status may occur:
   (a) The *token* reaches process P_i from process P_j (P_j is in the row just before P_i's row): Process P_i first updates *token.RTR* with its row number. Then, if there are any unresponded requests from nodes of Row m in $Waiting_i$, they will be recognized and inserted in *token.next* queue. Recognizing whether a *ReqMsg* is responded or not is done via *token.seqnum*. Now, if *token.next* is empty (there is not any unresponded request in this row), the *token* is sent to the next node below to continue its perpetual vertical path. But if *head*(*token.next*) is $ReqMsg(i, SN_i)$, $CS\text{-}permission_i$ is set to i and process P_i can enter its CS. Otherwise, the *token* is sent to a process in Row m, say process P_f, whose request is in the *head*(*token.next*). For details, see Lines 21 to 35 in Fig. 3.
   (b) When process P_i receives the *token* from process P_j (placed in P_i's row), process P_i sets $CS\text{-}permission_i$ to i, so that it can enter its CS. For details, see Lines 36 to 38 in Fig. 3.

Releasing the CS: Assume the status that process P_i wants to exit its CS. First, this process sets $CS\text{-}permission_i$ to zero, so that it cannot enter its CS repeatedly and causes starvation for other processes. $ReqMsg(i, SN_i)$ must be removed from *token.next* or $Waiting_i$, because the request of process P_i is responded. Process P_i updates *token.seqnum*[i] with SN_i and if there is a request in *head*(*token.next*), sends the *token* to the corresponding node. Otherwise, process P_i sends the *token* to it's *Down* to comply with the perpetual mobility of the *token*. For details, see Lines 9 to 16 in Fig. 3.

```
Initialization:
        token.seqnum[1...N]←0, token.RTR←a, token.next is empty.
        For all processes: SN←0, CS-permision←0, Waiting is empty.
Distributed Mutual Exclusion Solver:
1.  CASE REQUEST THE CS:
2.      $SN_i$←$SN_i$+1;
3.      CREATE ReqMsg(i,$SN_i$);
4.      INSERT (Waiting$_i$,ReqMsg(i,$SN_i$));
5.          /*inserts ReqMsg(i,$SN_i$) in the rear of Waiting$_i$.*/
6.      SEND ReqMsg(i,$SN_i$) to all nodes in row m except itself;
7.          /*multicasts ReqMsg(i,$SN_i$) to all nodes in its row.*/
8.      WHILE (CS-permision$_i$≠i);
9.  CASE RELEASE THE CS:
10.     CS-permision$_i$←0;
11.     REMOVE ReqMsg(i,$SN_i$) from head(token.next) or Waiting$_i$;
12.     token.seqnum[i]←$SN_i$;
13.     IF (token.next is empty) THEN SEND token to Down;
14.     ELSE
15.         EXTRACT head(token.next) which is ReqMsg(f,$SN_f$);
16.         SEND token to process $P_f$;
17. CASE RECEIVE A MESSAGE BY PROCESS $P_i$ FROM PROCESS $P_j$:
18.     SWITCH (message type)
19.         CASE ReqMsg(j,$SN_j$):
20.             INSERT (Waiting$_i$,ReqMsg(j,$SN_j$));
21.         CASE token:
22.             IF (token.RTR≠m) THEN token.RTR←m;
23.                 /*state in which token is received from a node in
24.                   previous row of process $P_i$ in the array.*/
25.                 WHILE (Waiting$_i$ is not empty)
26.                     REMOVE head(Waiting$_i$) which is ReqMsg(f,$SN_f$);
27.                     IF (token.seqnum[f]<$SN_f$) THEN
28.                         INSERT(token.next,ReqMsg(f,$SN_f$));
29.                 IF (token.next is empty) THEN SEND token to Down;
30.                     /*token continues its circular movement.*/
31.                 ELSE IF (head(token.next) is ReqMsg(i,$SN_i$)) THEN
32.                     CS-permision$_i$←i;
33.                 ELSE
34.                     EXTRACT head(token.next) which is ReqMsg(f,$SN_f$);
35.                     SEND token to process $P_f$;
36.             ELSE /*state in which the token reaches process $P_i$
37.                     because of $P_i$'s request.*/
38.                 CS-permision$_i$ ←i;
```

Fig. 3. Pseudo code of the new algorithm in process P_i.

5. Proof of the correctness

To ensure the correctness of the algorithm, it is sufficient to assure safety and liveness. Therefore, we must proof separately that these two basic needs are assured.

5.1. Safety is assured

A node must obtain the *token* to enter the CS, and release it only after exiting the CS. Since the *token* in the system is unique, safety is assured.

5.2. Liveness is assured

Liveness is assured if every request to enter the CS is eventually granted. Liveness implies freedom of deadlock and starvation.

Theorem 1 *(liveness). The algorithm in Fig.* 3 *confers liveness property.*

Proof. We prove this by contradiction, too. Therefore, suppose that the algorithm does not assure liveness. This assumption can be the result of the following situations:

1. None of the nodes is the token-holding node; therefore the *token* cannot be transferred to the other nodes: This assumption is incorrect because in the beginning of the algorithm, P_k is the token-holding process (based on the assumptions of the algorithm) and this *token* is sent from one node to another.
2. The token-holding node does not eventually get information about other nodes' requests: Process P_i in order to attempt to invoke its CS sends $ReqMsg(i, SN_i)$ directly to all nodes in its row except itself. These nodes, after receiving $ReqMsg(i, SN_i)$, insert it in their *Waiting*s. On the other hand, the *token* reaches one of these nodes, say process P_j, in the path of perpetual mobility of the *token*. When P_j becomes the token-holding process, it inserts $ReqMsg(i, SN_i)$ in the *token.next* (first stage). Note that, only when the *token* reaches process P_j in its vertical path, the existing unresponded *ReqMsg*s in $Waiting_j$ are inserted in *token.next*. After that, the *token* passes through nodes in P_j's row so that the existing *ReqMsg*s in *token.next* is served (second stage). Therefore, in this stage no insertion in *token.next* will occur. The new *ReqMsgs* from processes in P_j's row should wait until the *token* reaches them again in its vertical movement. As a result, there is no starvation for nodes in other rows because new *ReqMsg*s in a row do not keep the *token* in that row, forever. Therefore, the first assumption is incorrect.
3. The token-holding node keeps the *token* forever: If *ReqMsg* of the token-holding node exists in *head*(*token.next*), the token-holding node enters its CS and finishes executing its CS in a limited time. The token-holding node, after releasing its CS and removing its *ReqMsg* from *head*(*token.next*), becomes non-token-holding node in two cases:
   (a) If *token.next* becomes empty (since there is no insertion in *token.next* during the second stage), the token-holding node will pass the *token* to the next node below in the path of perpetual mobility of the *token* and becomes non-token-holding node.
   (b) If *token.next* does not become empty, the token-holding node will extract existing *ReqMsg* in *head* (*token.next*), suppose $ReqMsg(f, SN_f)$, and sends the *token* to process P_f then becomes non-token-holding node.

   This contradiction then shows that the anti-liveness assumption cannot be true.
4. Messages do not reach the destination node: Based on assumptions in the algorithm, network is error-free and nodes act correctly, thus this statement is incorrect, either.
5. Nodes' requests to enter their critical sections in *token.next* are unresponded: *token.next* is a FIFO queue without priority. Therefore, a node which its *ReqMsg* is inserted in *token.next*, eventually it receives the *token*. Therefore this assumption cannot be true, either.

In the end, liveness is assured. □

6. Performance and comparison

The execution time of instructions in the algorithm is assumed to be negligible, compared to the message transmission times. Hence, we focus on the number of message exchanges for performance evaluation. The performance of a DME algorithm is often studied under two special loading conditions, i.e., light load and heavy load.

6.1. Performance under light demand

Consider the light load situations, in which only process P_i attempts to invoke its CS. Therefore, process P_i sends $ReqMsg(i, SN_i)$ directly to all nodes in Row m except itself, $\sqrt{N} - 1$ message exchanges are required. All nodes in Row m except process P_i, after receiving $ReqMsg(i, SN_i)$, insert it in their *Waiting*s. On the other hand, from the time of inserting $ReqMsg(i, SN_i)$ in *Waiting* of nodes in Row m to the time that *token* in its perpetual mobility reaches one of the nodes in Row m, between 1 to $\sqrt{N}$ message exchanges are required. One message exchange is required when the token-holding node which is in the previous row sends the *token* in downward direction. $\sqrt{N}$ message exchanges are required when the token-holding node which is in Row m, without attending newly created request ($ReqMsg(i, SN_i)$) in the same row, sends the *token* in downward direction to continue perpetual mobility of the *token*. Finally, after that the *token* reaches one of the nodes in Row m, zero (receiver node of the *token* is process P_i) or one (receiver node of the *token* is not process P_i) message exchange is required for transferring the *token* to process P_i. So the algorithm in the worst case (under circumstance of existing at least one request in the system), requires $2\sqrt{N}$ message exchanges which are fewer than what are needed by similar algorithms [1,6–8,10,12,14,16] in this case. The number of overall message exchanges in the best case is $\sqrt{N}$, which is better than many similar algorithms [1,6,8,14] in the same case, too.

6.2. Performance under heavy demand

Suppose a heavy load situation, in which all nodes attempt to invoke their critical sections in time t_1 simultaneously and each node after releasing its CS, again attempts to invoke its CS immediately. As a result, each of N nodes sends its *ReqMsg* to all nodes in its row except itself (other $\sqrt{N} - 1$ nodes). So up to this step of the algorithm, $N(\sqrt{N} - 1)$ messages are exchanged totally. On the other hand, suppose the *token* is transferring from process P_i (placed in Row m and Column n) to process P_j (placed in Row $((m \bmod \sqrt{N}) + 1)$ and Column n) to continue the perpetual mobility of the *token* in Column n.

Assume that in time t_2, ($t_2 > t_1$), *ReqMsg*s of half of nodes in Row $((m \bmod \sqrt{N}) + 1)$, on average, reaches process P_j earlier than the *token*. When the *token* reaches process P_j, these *ReqMsg*s are appended to *token.next* (first stage). Then the *token* is passed from one node to another node in Row $((m \bmod \sqrt{N}) + 1)$ to satisfy the mentioned *ReqMsg*s (second stage). So, one message exchange is required for transferring the *token* to satisfy each of these *ReqMsg*s. Whenever *token.next* becomes empty, perpetual mobility of the *token* starts via transferring the *token* to downward direction. *ReqMsg*s of each row are responded similar to what mentioned above. Hence, at most $N/2 + \sqrt{N}$ messages are exchanged to transfer the *token* between $N/2$ requesting nodes. The same number of messages will be exchanged to satisfy the request of other nodes.

Generally $N + 2\sqrt{N} + N(\sqrt{N} - 1)$ messages are exchanged in this situation. Thus, the number of message exchanges per CS invocation in the average case of heavy demand situation is:

$$\bigl(N + 2\sqrt{N} + N\bigl(\sqrt{N} - 1\bigr)\bigr)/N \simeq \sqrt{N}.$$

Which is better than many other algorithms (e.g. [1,6,8,14, 16]) in the same case.

7. Discussion

The perpetual mobility of the *token* in very light load situations wastes resources of the DS. It seems to be a disadvantage of the algorithm because the *token* sometimes circulates uselessly. Therefore, we recommend that the algorithm not be implemented on the distributed systems with very light demand. On the other hand, circulating of the *token* in a special column imposes extra load on nodes in that column. To prevent the extra load, changing column can be done if there is not any request in the circular path of the *token*.

There are two applicable aspects of the algorithm which can be discussed here. In the first subsection, considering different dimensions instead of current equal dimensions, the logical topology is expanded. The following subsection represents a preliminary step in giving a solution to a challenging problem, DME in mobile ad hoc networks.

7.1. Expanding the logical topology

The proposed algorithm can be implemented by a logical two-dimensional array with u rows and v columns. In this case, process P_i sends its *ReqMsg* to enter the CS to all other nodes in its row using $v - 1$ message exchanges. As it was mentioned in Section 6, for the arrival of the *token* at process P_i, between 1 to $u + 1$ message exchanges are needed. As a consequence, v message exchanges in the best case and $v + u$ message exchanges in the worst case are required.

If we assume that $v = 1$, then the algorithm will become similar to the token-ring algorithm [7]. One message exchange in the best case and u (or N) message exchanges in the worst case are required. If we assume that $u = 1$, then the algorithm becomes similar to Suzuki–Kasami's algorithm [16]. As a result, v (or N) message exchanges in both best and worst cases are needed.

Therefore, we can change the dimensions of the proposed topology with respect to different applications. Of course, the number of message exchanges will be changed, too.

7.2. Expanding the algorithm to new application

Recently, the mutual exclusion problem received an interest for mobile ad hoc networks [3,4,9,18,20]. To the best of our knowledge, few algorithms have been proposed in the literature and all of them are token-based approaches.

Consider a clustered ad hoc network. Some limitations for this network are: the number of nodes in each cluster is constant, and, within a cluster, nodes have low mobility. The algorithm can be mapped into some applications of ad hoc networks for which the number of message exchanges under heavy load conditions is very important. To map, it is sufficient to consider nodes in each row as members of a cluster. Therefore, each node sends its CS entry request to other members in its cluster and waits until it receives the *token*. On the other hand, the *token* circulates between clusters and satisfies request of nodes. Note that v is assumed to be the number of columns (or cluster size) and u is assumed to be the number of rows (or clusters). Similar to what is mentioned in Section 6.2, the number of message exchanges for requests is $v(v - 1)$ in each cluster and $vu(v - 1)$ in all clusters. The number of message exchanges for the token movement is $v - 1$ in each cluster and $u(v - 1)$ in all clusters. Therefore, there are $vu(v-1)+u(v-1) = u(v^2-1)$ message exchanges intra clusters, totally. The number of message exchanges required for the inter cluster *token* movement is $2u$. Now, if we consider the energy of intra cluster message exchanges as E_c and the energy of inter cluster message exchanges as E_x, we can conclude that energy required for each node is intermediately

$$\bigl(u\bigl(v^2 - 1\bigr)E_c + 2uE_x\bigr)/uv \simeq vE_c + (2/v)E_x.$$

This relation shows that with a constant number of nodes, the more the number of intra cluster nodes (the number of columns) increases, the more the wasted energy for inter cluster message exchanges decreases. As a result, depending on the application (ratio of E_x with E_c), the optimal value for v can be gained so that it can decrease energy consumption and consequently increase network lifetime.

8. Conclusion

The proposed hybrid token-based DME algorithm uses both the perpetual mobility of the token and the token-asking method. Therefore, we assumed a logical topology in the form of wraparound two-dimensional array, which applies the concept of perpetual mobility of the token in columns and concept of token-asking in rows. We proved that this algorithm satisfies the requests of entering the critical sections, correctly. Therefore, safety and liveness properties are assured.

We have increase in scalability property and decrease in average waiting time and also the overhead due to additional communication per CS invocation in comparison with many other algorithms.

Generally, in light demand conditions, the number of necessary message exchanges, $2\sqrt{N}$, is more than that of the heavy demand conditions, $\sqrt{N}$, per CS invocation. The performance of the algorithm is better in comparison with many other algorithms and requires fewer message exchanges, especially in the heavy load situations or in applications in which upper bound of the number of message exchanges is important. For the future works, we will focus on DME in mobile ad hoc networks.

References

[1] Md. Abdur Razzaque, C. Seon Hong, Multi-token distributed mutual exclusion algorithm, in: 22nd International Conference on Advanced Information Networking and Applications, March 2008, pp. 963–970.

[2] D. Agrawal, A. El Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, ACM Transactions on Computer Systems 9 (1) (February 1991) 1–20.

[3] R. Baldoni, A. Virgillito, R. Petrassi, A distributed mutual exclusion algorithm for mobile Ad-Hoc networks, in: Proceeding of the Seventh International symposium on Computers and Communications, (ISCC'02), November 2002, pp. 539–544.

[4] M. Benchaïba, A. Bouabdallah, N. Badache, M. Ahmed-Nacer, Distributed mutual exclusion algorithms in mobile Ad hoc networks: An overview, ACM SIGOPS Operating Systems Review 38 (1) (January 2004) 74–89.

[5] E.W. Dijkstra, Solution of a problem in concurrent programming control, Communication of the ACM 8 (9) (September 1965) 569.

[6] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communication of the ACM 21 (7) (July 1978) 558–565.

[7] G. Le Lann, Distributed systems towards of a formal approach, in: IFIP Congress, North-Holland, 1977, pp. 155–160.

[8] M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, ACM Transactions on Computer Systems 3 (2) (May 1985) 145–159.

[9] M. Moallemi, M.H. Yaghmaee Moghaddam, M. Naghibzadeh, A fault-tolerant mutual exclusion resource reservation protocol for clustered mobile ad hoc networks, in: 8th International Conference on ACIS, Qingdao, July 2007, pp. 528–533.

[10] M. Naimi, M. Trehel, A. Arnold, A $\log(n)$ distributed mutual exclusion algorithm based on the path reversal, J. Parallel and Distributed Computing 34 (1) (April 1996) 1–13.

[11] S. Paydar, M. Naghibzadeh, A. Yavari, A hybrid distributed mutual exclusion algorithm, in: 2nd International Conference on Emerging Technologies, 13–14 November 2006, pp. 263–270.

[12] K. Raymond, A tree-based algorithm for distributed mutual exclusion, ACM Transactions on Computer Systems 7 (1) (February 1989) 61–77.

[13] M. Raynal, A simple taxonomy for distributed mutual exclusion algorithms, in: Operating Systems Review, ACM Press, 1991, pp. 47–49.

[14] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Communication of the ACM 24 (1) (January 1981) 9–17.

[15] P.C. Saxena, J. Rai, A survey of permission-based distributed mutual exclusion algorithms, Computer Standards and Interfaces 25 (2003) 159–181.

[16] I. Suzuki, T. Kasami, A distributed mutual exclusion algorithm, ACM Transactions on Computer Systems 3 (4) (November 1985) 344–349.

[17] A.S. Tanenbaum, M.V. Steen, Distributed Systems Principles and Paradigms, 2nd edition, Prentice-Hall International, 2007.

[18] R. Vedantham, Z. Zhuang, R. Sivakumar, Mutual exclusion in wireless sensor and actor networks, in: Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), Reston, VA, USA, September 2006, pp. 346–355.

[19] M. Velazquez, A survey of distributed mutual exclusion algorithms, Technical Report CS-93-116, Colorado State University, September 1993.

[20] W. Zheng, L. Xin Song, L. Meian, Ad hoc distributed mutual exclusion algorithm based on token-asking, J. Systems Engineering and Electronics 18 (2) (2007) 398–406.