Info-based Approach in Distributed Mutual Exclusion Algorithms

Peyman Neamatollahia, Hoda Taherib, and Mahmoud Naghibzadehc

Abstract: In this paper, we propose a token-based fully distributed algorithm with token-asking method for Distributed Mutual Exclusion (DME) in a computer network composed of $N$ nodes that communicate by message exchanges. The main goal is to introduce a new class of token-based DME algorithms called info-based. In some previous algorithms, the request to enter a critical section is sent to all nodes because the token-holding node is unknown, but in this info-based algorithm some nodes know the token-holding node and lead critical section entering requests to it, directly. This algorithm uses a logical structure in the form of a wraparound two-dimensional array which is imposed on the interconnecting network. Usually, a request message for entering the critical section is sent vertically down in the array, and eventually sent to the token-holding node with the assistant of an informed-node (common node between the row consisting of the token-holding node and the column consisting of the requester node). The nodes invoking the critical section can obtain the token with fewer message exchanges in comparison with many other algorithms. Typically, the number of message exchanges is $4\sqrt{N} + 1$ under light demand which reduces to approximately 2 message exchanges under heavy demand. A correctness proof is provided.

Keywords: Mutual exclusion, critical section, distributed systems, token-based algorithm, message exchange, concurrency.

This is the pre-print version of the paper. The final publication is available at Elsevier via http://dx.doi.org/10.1016/j.ipdc.2012.01.005.
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. A single computer can also be viewed as a DS in which the central control unit, the memory units, and the input-output channels are separate processes [1].

One of the most important purposes of the distributed systems is to provide an efficient and convenient environment for sharing resources [2]. 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. An important application of distributed systems, which uses mutual exclusion and needs special mention, is in the field of replicated databases. A replicated database is a distributed database in which some data items are stored redundantly at multiple sites. If a node is to perform updates, it must ensure that no other node is doing this activity. All replica control protocols require that mutual exclusion must be guaranteed between two write operations or a read and write operation [2]. When a process has to read or update certain shared data structures, it first enters a CS to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time [3]. Obtaining dedicated access to a resource is a basic problem in DS. If a resource needs to be accessed exclusively, Mutual Exclusion (ME), some controls are necessary for assuring 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. The problem of DME has to be solved to prevent race condition and, as a result, prevent the possibility of producing an incorrect result by a correct program [4]. In a DS any given node has only a partial or incomplete view of the total system [5]. In DS none of the shared variables, semaphores or local kernel methods can be used for implementing the DME. 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 its CS from among other nodes that are attempting to invoke their critical sections.

We attempt to propose a new token-based algorithm for solving the DME problem. In our algorithm, the token moves from one idle token-holding node (i.e. the node that has the token but is not interested in the CS) to the requesting node, based on the arriving request from the requesting node. We used a wrap around two-dimensional array logical topology to decrease the number of message exchanges. Usually, a request message for entering the CS is sent vertically down in the array. The role of the common node between the row consisting of the token-holding node and the column consisting of requester node is to send the CS entry requests directly to the token-holding node. The nodes which know the token-holding node are named informed-nodes. Eventually, the request message for entering the CS is sent to the token-holding node with the assistant of informed-nodes.

The rest of paper is organized as follows: Section 2 gives a small survey of DME algorithms and splits these algorithms into two families of algorithms (token-based and non-token-based algorithms), Section 3 considers assumptions for the algorithm, Section 4 considers the new algorithm by explaining the main idea, control messages and data structures, and then the description of algorithm (the pseudo-code of the algorithm is shown). This section also describes details of the algorithm with a scenario, Section 5 proves the correctness of the algorithm (safety and liveness properties), Section 6 calculates the performance of the algorithm and then compares it with performance of other algorithms in a table, Section 7 presents the simulation results, Section 8 discusses about the logical topology and conclusion is at the end.

2 RELATED WORK

Solving the ME problem (which is first introduced by Dijkstra [6]) has been one of the topics which has received the attentions of many researchers. As there are not enough resources to fulfill the requirements of all concurrent processes in multiprogramming, multiprocessor, and distributed systems, resources are shared by all processes. In spite of many algorithms in the DME concept, only few have been very innovative and presenting new ideas or new algorithmic techniques. Two general approaches for solving DME problem, are centralized and distributed approaches. In a centralized approach, one process is considered as a coordinator process which is the controller for a shared object. Whenever a process wants to access the shared object, it sends a request message to the coordinator process and when the shared object becomes available and the request’s turn pops up, the coordinator process returns a reply message which means that the shared object is free. Of course, this unique reply can be implemented by a token which is managed by the coordinator process. Hence, this approach can be the intersection point between two concepts, token-based and permission-based; these concepts will be explained later. But there are problems such as coordinator process becoming a bottleneck and having single point of failure. In distributed approach, there are two families of algorithms which are token-based and non-token-based algorithms. The schema in Fig. 1 shows a more fine grained classification of DME algorithms.

2.1 Token-based Algorithms

In these 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 doing 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 only one process at the 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 is 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 [7].
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 very few numbers of processes attempting to invoke their critical sections, simultaneously. But the perpetual mobility of the token is very effective on the high load situations. Token-ring algorithm [8] is one of these algorithms. In this algorithm in order not to forget the request of some processes, they are put on a directed logical ring and the token moves, clockwise or counter clockwise, around the logical ring. If a process receives the token and is not interested in its CS, it passes the token to the next process along the ring. The perpetual mobility on a unidirectional ring ensures the liveness. In addition to the perpetual mobility of the token, the other 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 method, a process which is attempting to invoke its CS, if it is not the token-holding process, requests for receiving 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 it the token. 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 [9] presented an algorithm that process \( P_i \) which is attempting to invoke its CS, broadcasts a request 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. Some of the algorithms of the token-asking method organize processes in the logical tree structure. These structures are classified to static and dynamic ones. For process \( P_i \), define \( \text{NEIGHBOR}_i \), to be the set of processes \( P_j \) such that there is an edge from process \( P_i \) to process \( P_j \) or from process \( P_j \) to process \( P_i \). In a dynamic structure, \( \text{NEIGHBOR} \) changes from time to time for each process \( P_i \). On the other hand, in a static structure, \( \text{NEIGHBOR} \) is constant for every process \( P_i \) in the system. Raymond [10] presented an algorithm based on static logical unrooted tree structure. The performance of this algorithm depends on the precise structure of the network’s spanning tree used, but the average number of message exchanges per CS invocation is \( O(\log N) \) and \( 2(N - 1) \) in the worst case. However, the worst case scenario usually does not occur frequently. This algorithm uses a surrogate mechanism in which a process \( P_i \), requests another process \( P_j \) to act on \( P_i \)'s behalf. Therefore, CS entering requests are indirectly led to the token-holding process and the token is also sent from the token-holding process to the requesting process indirectly. This concept is similar to info-based, but in our info-based algorithm CS entering requests are directly led to the token-holding process and the token is also sent from the token-holding process to the requesting process directly. In the tree structure of Raymond's algorithm, non-leaf processes should tolerate more workload than leaf processes. But in our algorithm, all processes receive approximately equal workload. Naimi et al [11] presented an algorithm based on dynamic logical rooted tree structure which requires \( O(\log N) \) message exchanges per CS invocation in an average case and \( N \) message exchanges in the worst case. In this algorithm CS entering requests are also led to the token-holding process indirectly. The superiority of the proposed algorithm over these two algorithms is demonstrated in forthcoming sections.

Also, it is possible to combine the two mentioned techniques. For instance, the authors in [12] presented a hybrid method. This method applies the concept of perpetual mobility of the token in columns and token-asking in rows of the torus logical topology. Fair scheduling of the token among competing processes is the major design issue of the token-based DME algorithms.

### 2.2 Non-token-based Algorithms

Lamport [1] presented an algorithm that requires \( 3(N - 1) \) message exchanges per CS invocation. A large classification of non-token-based algorithms includes permission-based algorithms. The idea is very simple: when a process attempts to invoke its CS, it asks the other processes whether they allow it to enter its CS or not, then it waits until all replies arrive. Of course, by this method a priority should be considered between two processes which have conflicting requests. Ricart and Agrawala [13] presented an algorithm which requires \( 2(N - 1) \) message exchanges per CS invocation. In the multi-token method, which is presented by Abdur Razzaque and Seon Hong [14], if process \( P_i \) attempting to invoke its CS, creates the \( \text{token}(SN_{i,j}) \) and passes it to next process in the proposed unidirectional logical ring and waits until it receives that message again, then enters its CS (here, the tokens act as permissions). If process \( P_i \) has already created a token, \( \text{token}(SN_{i,j}) \), with a higher priority than \( \text{token}(SN_{i,i}) \), it means \( SN_j < SN_i \) if \( SN_j = SN_i \), then \( j < i \), it avoids sending \( \text{token}(SN_{i,j}) \) to the next process in the unidirectional logical ring until executing its CS. This algorithm requires \( N \) message exchanges per CS invocation. Disadvantage of permission-based algorithms is the high overhead in their communications in comparison with the token-based algorithms.

To decrease the number of message exchanges, some algorithms propose that the process attempting to invoke its CS, need not get permission from all other processes, but rather from a subset of processes. A part of these algorithms partition processes into a collection of subsets, such that every pair of subsets has at least one shared process i.e. the intersection of every pair of subsets is not empty. These kinds of subsets and the corresponding ME solver algorithms are named quorums and quorum-based algorithms, respectively. Maekawa [15] presented the first quorum-based algorithm: process \( P_i \) attempting to invoke its CS requires blocking other processes in the same quorum from entering their critical sections to use the same resource. However, it will unblock these processes whenever it is finished with the resource. Considering even special messages used for preventing deadlock, this algorithm dramatically reduces the number of message exchanges to \( c(\sqrt{N} - 1) \), where \( c \) is an integer with \( 3 \leq c \leq 5 \). This new thought helps us to choose a good topology for our proposed algorithm. Many other algorithms (for example [16, 17, 18, 19, 20, 21, 22, and 23]) exist that use quorums in order to reduce the message complexity. Agrawal and Abbadi [16] presented an algorithm which uses a logical tree topology for creating quorums. It decreases the number of message exchanges to \( O(\log N) \) in the average case, but requires \( (N + 1)/2 \) message exchanges in the worst case. Cao and Singhal [17] presented an algorithm which
limits the number of message exchanges to 3k under light demand and 6k under heavy demand. Here, \( k = \sqrt[3]{N} \) if Maekawa’s quorum construction algorithm is used and it is \( \log N \) if Agrawal and Abbadi’s quorum construction algorithm is used. Quorum-based algorithms, in comparison with other permission-based algorithms, have lower number of message exchanges. However, the problem of recreating quorums and managing them is the disadvantage of this type of algorithms.

Also, there are algorithms that try to use a combination of token-based and permission-based concepts. One kind of these hybrid algorithms is presented by Paydar et al [4]. They used a two-dimensional array logical topology with quorum-based concept. It is one of the token-based algorithms. With this method, the number of message exchanges is \( 4\sqrt[3]{N} \), in the worst case.

### 3 Assumptions

What we will present in this paper solves ME problem in a DS composed of \( N \) nodes with no shared memory. These nodes communicate through asynchronous message passing on a communication network layer that is assumed to be error-free. At first, without loss of generality, we assume that there is only one process in each node. However, it is possible that more than one process, which are interested in entering their critical sections, could exist in every node. Therefore, we use process and node to represent the same concept. We assume that for any two processes \( P_i \) and \( P_j \), the messages sent from process \( P_i \) to process \( P_j \) are received in the same order as they are sent. Message propagation delay is unpredictable but it is finite, it indicates that every message will eventually be received. This assumption avoids introducing message acknowledgement protocols.

The algorithm does not entail any specific physical interconnection topology. In other words, the physical topology of the network is known (e.g. ring, star, mesh, etc). Therefore, every process can send messages to all other processes (complete communication graph similar to [1, 4, 9, 10, 11, 13, and 16]). Because the requests move on a vertical ring in down direction and on the other hand, the informed-nodes locate on the horizontal ring, wraparound two-dimensional array (i.e. 2-D torus) is selected for the logical structure of the interconnecting network. This logical topology is only a conceptual tool used to describe the algorithm (similar to [4, 12, and 18]). It is assumed that \( N = d^2 \) where \( d \) is an integer and \( N \) is the number of processes. Therefore, the logical interconnecting array is composed of \( \sqrt[3]{N} \) rows and \( \sqrt[3]{N} \) columns.

Initially, a unique identification number between 1 to \( N \) is randomly assigned to each process. Each process (say \( P_i \)) computes its row and column numbers in the wraparound two-dimensional array (for example \( P_i \)'s row is equal to \( \lceil i/d \rceil \)). Besides, each node knows other nodes located in its row \( (k = \lceil i/d \rceil, \forall j, (k-1)d < j \leq kd) \) and also its down neighbor (see Fig. 2).

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 CS entering requests might not be satisfied in the order of their construct, like algorithms proposed in [5, 6, 7, 8, 9, 10, 11, and 13].

### 4 An Info-based Algorithm

The explanation of the algorithm is split into three parts: first, the main idea is described, second, control messages and data structures are introduced and third, the overall algorithm is presented. At the end of this section, a scenario is declared.

#### 4.1 The Main Idea

By info-based we mean that from total nodes in a DS, some nodes know the current location of the token and forward CS entry requests to the token-holding node, directly. Therefore, in our info-based algorithm, CS entering requests are led to the token-holding process directly through informed-nodes and the token is also sent from the token-holding process to the requesting process directly. As a result, knowing that some nodes have requests to enter their critical sections and the current location of the token are very important concepts in our info-based algorithm.

We assume that in the beginning of the algorithm, \( P_0 \) is the token-holding process (which is in row \( a \) and column \( b \) of the wraparound topology) and it is executing its CS. To simplify, assume there is only one non-token-holding process, say process \( P_i \), in row \( m \) and column \( n \), 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. 3. The request message (\( ReqMsg \)) of process \( P_i \) for entering its CS is sent to the node below \( P_i \), suppose it is process \( P_j \). If process \( P_j \) does not know who the token-holding node is, it sends \( ReqMsg \) of process \( P_0 \) to the next node below. This action continues until \( ReqMsg \) of process \( P_0 \) eventually arrives to one of the informed-nodes, a node in row \( a \) that knows process \( P_0 \) is the token-holding node, e.g. process \( P_k \) in Fig. 3. Now, \( ReqMsg \) of process \( P_0 \) is sent directly to process \( P_k \). Therefore, up to this step of the algorithm, the request message of process \( P_i \) for entering its CS has arrived to the token-holding node. Process \( P_k \) after releasing its CS, sends the token directly to process \( P_i \).

Process \( P_i \) after receiving the \( token \), informs all the nodes in its row that \( P_i \) is the token-holding process by an \( InfoMsg \) message. Hence, all nodes in row \( m \) know that \( P_i \) is the token-holding process. In this case, process \( P_i \) through sending \( RelMsg \) message to process \( P_k \), asks process \( P_k \) to inform all nodes in \( P_i \)'s row that \( P_i \) does not hold the \( token \) anymore.

To do this, process \( P_i \) after receiving the \( RelMsg \) message, multicasts \( RowRel \) messages to all nodes in its row, except itself. Process \( P_i \) waits until receiving \( Ack \) messages from all these nodes. By arriving each \( Ack \) message from any process (say process \( P_j \), process \( P_j \) knows for certain that it has received all \( ReqMsgs \) sent through process \( P_i \) already. When process \( P_j \) receives the \( Ack \) messages from all these nodes (i.e. all nodes in \( P_i \)'s row know that \( P_i \) is a non-token-holding process anymore), it sends \( Finished \) message to process \( P_i \). Therefore, the responsibility of process \( P_k \) in managing the \( token \) and CS entering requests is finished.
Process \( P \), after receiving the *Finished* message, executes its CS and after completing its CS becomes the manager of the *token* with the power to decide whether remain the idle token-holding process or send the *token* to another node.

In our algorithm, there is always only one row which all nodes in it are the informed-nodes and these nodes remain informed-nodes until another row appears that includes the informed-nodes. On the other hand, the request of each process is sent vertically in down direction until arrives to one of the informed-nodes.

To decrease the number of message exchanges, we used two interesting principles in the algorithm which have considerable influence on the overall performance:

- **PCL-1**: when a process (say process \( P_i \)) is waiting to receive the *token*, if it receives a request from another process (say process \( P_j \)) in its column, it blocks that request. Whenever process \( P_i \) receives the *token* and executes its CS, it inserts that request into the *token* queue of not responded requests. Since process \( P_i \) will eventually get the *token* and has the knowledge about the request of process \( P_j \), there is no need for that request to continue its vertical path in order to arrive to one of the informed-nodes.

- **PCL-2**: As mentioned earlier, we name a node that holds the *token* the token-holding node. There are two types of token-holding nodes: explicit and the implicit token-holding node. By explicit we mean that the token-holding node informs all nodes in its row that it has the *token*, before executing its CS. Therefore, these nodes become the informed-nodes. In contrast to an explicit token-holding node, an implicit token-holding node executes its CS, without informing nodes in its row. Consider the following conditions. (1) Process \( P_i \) has received the *token* but has not entered its CS, yet. (2) Except the request of process \( P_i \), other not responded requests exist in the *token* queue of not responded requests. Under the assuming two, process \( P_i \) enters its CS without informing the nodes in its row that it is the token-holding process now. Therefore, \( P_i \) is the implicit token-holding process. The reason is that process \( P_i \) holds the *token* temporarily and after its request is satisfied; it must pass the *token* to the next requester node, immediately.

### 4.2 Control Messages and Data Structures

In this section, we describe the control messages and data structures used in the algorithm through Fig. 4. The following local data structures are used by process \( P_i \):

- **\( SN_i \)**: a counter that process \( P_i \) increases by one whenever it attempts to invoke its CS, to indicate that there is a request from this process which is not responded.
- **\( Waiting_i \)**: a FIFO queue which is composed of not responded \( ReqMsgs \).
- **\( P_i.waiting \)**: a FIFO queue which is composed of \( ReqMsgs \) in which some requests are responded and some are deferred.
- **\( CL_token \)**: a variable to keep the node identifier of the current explicit token-holding node (useful for informed-nodes). Another use of this variable is that if \( P_i \) is the token-holding process, it can enter its CS when \( CL_token \) equals to \( i \).
- **\( SL_token \)**: a variable that shows the beginning and the end of the responsibility of the explicit/implicit token-holding process after the "token". The beginning is shown by \( SL_token_{\text{end}} \), and the end by \( SL_token_{\text{end}}=0 \).
- **\( Right \) and \( Down \)**: every node knows its right and down neighbors which are represented by constant identifiers placed in Right and Down variables, respectively.
- **\( Q \)**: a temporary FIFO queue of \( ReqMsgs \) which are probably deferred.
- **\( NumAck \)**: the number of acknowledgment messages that have received by process \( P_i \).
- \( ReqMsgs, InfoMsgs, RelMsgs, RowRel, Ack \) and *Finished* are considered message types which are used in the algorithm and *token* is a record which is sent by a message. The details of these messages follow:

- **\( token \)**: this is a record composed of an array with \( N \) elements named *seqnum*\(^1\) (similar to [9] for distinguishing responded requests from not responded ones), a FIFO queue including not yet responded requests named *next*, a FIFO queue including requests which are probably deferred named *testreq* and two variables named *idexp* and *rowexp*. The *token.idexp* and the *token.rowexp* show identification number of the explicit token-holding node and its row, respectively.

- **\( ReqMsgs(i,SN) \)**: a message which is sent by process \( P_i \) to invoke its CS. It is composed of the identification number of the process, \( i \), and its sequence number, \( SN \), which is shown by \( ReqMsgs(i,SN) \).

- **\( InfoMsgs(i,Q) \)**: this message informs the receiving node that process \( P_i \) is the explicit token-holding node. On the other hand, \( Q \) is a queue in which \( ReqMsgs \) are saved.

- **\( RelMsgs(i,sw) \)**: this message requests from its receiver node to inform all nodes in its row that it is not the token-holding node, any more. "sw" is a boolean parameter and is set to true when the sender and receiver of this message are in the same row, otherwise it is false.

- **\( RowRel(i,l) \)**: this message informs the receiver node that process \( P_i \) is not the token-holding node any more. Parameter "\( l \)" is set to zero if the new and the old token-holding nodes are not in the same row. Otherwise, it is set with the identification number of the new token-holding node.

- **\( Ack(i) \)**: this is an acknowledge message which is sent by process \( P_i \).

- **\( Finished(\text{Waiting}) \)**: this message means that the sender process, process \( P_i \), has finished its job. By this message, *Waiting* is sent, too.

For the following section we assume that node \( k \), \( k \) is a constant, in row \( a \) (\( 1 \leq a \leq \sqrt{N} \)) and column \( b \) (\( 1 \leq b \leq \sqrt{N} \)) is the

\(^1\) It is an array with \( N \) elements and *seqnum[i]* indicates how many times process \( P_i \) has entered its CS until now.
We investigate the behavior of the algorithm in three cases (1) process \( P_i \) requests entering its CS, (2) process \( P_i \) receives a message from process \( P_j \), and (3) process \( P_i \) leaves its CS.

**Request the CS:** Process \( P_i \) to execute its CS, first increases its sequence number, \( S_N \), by one. It then creates a request in the form of \( ReqMsg(i,SN) \) and inserts a copy of it in the tail of its \( Waiting \) queue, \( Waiting \). Now if process \( P_i \) is not the token-holding node but knows which node is the token-holding one (i.e., process \( P_i \) is an informed-node), sends \( ReqMsg(i,SN) \) to the explicit token-holding node. Otherwise, if process \( P_i \) is neither a token-holding node nor an informed-node, inserts a copy of \( ReqMsg(i,SN) \) in \( P_Waiting \), and sends \( ReqMsg(i,SN) \) to its \( Down \). Therefore, \( ReqMsg(i,SN) \) starts its vertical movement until it arrives to one of the informed-nodes. Conditional statements mentioned above are done atomically. Then, process \( P_i \) waits for receiving the \( token \). After receiving, it waits until \( ReqMsg(i,SN) \) comes to \( head(token.next) \) and \( CL_token \) is equal to \( i \). This means that all conditions for process \( P_i \) are met to execute its CS. Process \( P_i \), before entering its CS, updates \( SL_token \) with \( i \) and then begins executing its CS. For details see Fig. 5.

**Receive a message:** When process \( P_i \) receives a message from process \( P_j \), depending on the type of message received, seven situations are possible:

1) \( ReqMsg(j,SN) \): This part of the algorithm must be done atomically. By receiving \( ReqMsg(j,SN) \), some cases may occur:
   - If process \( P_i \) is a token-holding node, it inserts \( ReqMsg(j,SN) \) in the \( Waiting \).
   - Otherwise, if \( Waiting \) is not empty (i.e., \( ReqMsg(j,SN) \) exist in the \( Waiting \) and process \( P_i \) waits to receive the \( token \), \( ReqMsg(j,SN) \) is inserted in \( Waiting \), but there is no need to send \( ReqMsg(j,SN) \) to the next nodes in vertical path (PCL-1).
   - Otherwise, if process \( P_i \) is an informed-node, it sends \( ReqMsg(j,SN) \) to the explicit token-holding node.
   - Otherwise, if \( ReqMsg(j,SN) \) has not arrived to process \( P_i \) itself, during its vertical movement, process \( P_i \) inserts \( ReqMsg(j,SN) \) in \( P_Waiting \) and sends it to its \( Down \). These conditions force a request to move forward within the column where it either arrives to an informed-node or it visits all the nodes in the corresponding column (see Case 1 in Fig. 6).

2) The \( token \): In this case, at first, process \( P_i \) appends \( Waiting \) to \( token.next \). Now, if there are any \( ReqMsgs \) except \( ReqMsg(i,SN) \) in \( token.next \), \( P_i \) becomes the implicit token-holding process (PCL-2). It means process \( P_i \) without informing nodes in its row that it is the token-holding node, sets \( CL_token \) to \( i \) and enters its CS. This principle (PCL-2) causes considerable decrease in the number of message exchanges for this case. If only \( ReqMsg(i,SN) \) exists in \( token.next \), it is necessary that process \( P_i \) informs nodes in its row that it holds the \( token \) before executing its CS. Therefore, process \( P_i \), in order to become an explicit token-holding node, sets \( SL_token \) to \( i \), then creates a queue named \( Q \) and appends \( P_Waiting \) to \( Q \). After that, process \( P_i \) creates \( InfoMsg(j,Q) \) and sends it to its \( Right \). The goal of process \( P_i \) is that besides informing all nodes in its row about the explicit token-holding node’s identification number, all possible pending \( ReqMsgs \) in \( P_waiting \) of the mentioned nodes can be collected (see Case 2 in Fig. 6).

3) \( InfoMsg(j,Q) \): In this case, process \( P_i \) in order to inform all nodes in its row (one of these nodes is process \( P_i \) itself) that it is an explicit token-holding node, sends \( InfoMsg(j,Q) \) to its \( Right \). Any receiver node also sends \( InfoMsg(j,Q) \) to its \( Right \) and this procedure continues until \( InfoMsg(j,Q) \) arrives to process \( P_i \). Now one of the following two states may occur:
   - If process \( P_i \) does not hold the \( token \), first it sets \( CL_token \) to \( j \). So, process \( P_i \) knows that process \( P_j \) is the token-holding node from now on and becomes an informed-node. Then process \( P_i \) appends \( P_Waiting \) to \( Q \) and sends \( InfoMsg(j,Q) \) to its \( Right \).
   - If \( InfoMsg(j,Q) \) has arrived back to its creator \((i=j)\), there are some \( ReqMsgs \) which are aggregated in \( Q \). Now process \( P_i \) appends \( Q \) to \( token.testreq \), then releases the occupied space of \( Q \). Now, if the old explicit token-holding node is in the same row of process \( P_i \), process \( P_i \) sends \( RelMsg(j,true) \) to that process. Otherwise, process \( P_i \) sends \( RelMsg(j,false) \) to it. For details see Case 3 in Fig. 6.

4) \( RelMsg(j,sw) \): In this case, process \( P_i \) is the old explicit token-holding node. By receiving this message, process \( P_i \) must inform all nodes in its row that it does not hold the \( token \), any more. On the other hand, it should collect all \( ReqMsgs \) received by nodes in its row, up to this time. Therefore, if process \( P_i \) and process \( P_j \) are in the same row \((sw=true)\), process \( P_i \) broadcasts \( RowRel(i,j) \) to all nodes in its row. Otherwise, it broadcasts \( RowRel(i,0) \) to all nodes in the row. Then it waits until \( Ack \)s from all nodes in its row arrive. After receiving these acknowledges, process \( P_i \) updates \( CL_token \) and \( SL_token \), and sends \( Finished \) in company with \( Waiting \) to the new explicit token-holding node (see Case 4 in Fig. 6).

5) \( RowRel(j,1) \): Process \( P_i \) by receiving this message, updates its information about the token-holding node and then sends \( Ack(i) \) to process \( P_j \) (see Case 5 in Fig. 6).

6) \( Ack(j) \): Receiving this message by process \( P_i \) means that, firstly, the sender of the message (process \( P_j \)) has updated its information about the token-holding node. Secondly, process \( P_i \) is sure to have received all \( ReqMsgs \) that are in the way from
process $P_j$ to itself (because we mentioned in Section 3 that requests are received in the order of their sent). Process $P_i$ by receiving this message increases the number of received Acks by one (see Case 6 in Fig. 6).

7) **Finished(Waiting):** This message means that process $P_i$ has finished its work as the old explicit token-holding node. In this case, process $P_i$ first appends Waiting to token.next. Then by comparing the $SN$ of eachReqMsg with the same element in token.seqnum, recognizes all pending ReqMsgs and convey them from token.testreq to token.next. Then process $P_e$ after updating token.rowexp and token.idexp, sets CL_token, to i and enters its CS (see Case 7 in Fig. 6).

**Release the CS:** When process $P_i$ finishes executing its CS, it sets CL_token, to zero which means it no longer possesses the token. Now, if any ReqMsg(i,SN) exists in token.next or Waiting queues, it removes them. Process $P_i$ updates token.seqnum to SN. If process $P_i$ is the explicit token-holding node and there is not any ReqMsg in token.next, it must wait until receiving a ReqMsg. In the case that a ReqMsg arrives to process $P_i$ from another node or there was a ReqMsg in token.next previously, process $P_i$ must append Waiting to token.next before sending the token to the requester node. Then process $P_i$ can send the token to it. If process $P_i$ is the implicit token-holding node, it is sufficient to set SL_token, to zero, append Waiting to token.next, and send the token to the next requester node (see Fig. 7).

### 4.4 A Scenario

In addition to describing a pseudo-code for formal presentation of the algorithm, we tried to consider a scenario for explaining the details. As it is shown in Fig. 8(a), $P_{16}$ is the explicit token-holding process.

In time $t_1$ while process $P_{16}$ is executing its CS, process $P_{19}$ attempts to invoke its CS, therefore increases variable $SN_{19}$ by one and inserts ReqMsg(19,SN) in Waiting, and after that, sends ReqMsg(19,SN) to process $P_{16}$ directly, in Fig. 8(b), the reason of sending the message directly is that process $P_i$ is an informed-node. Process $P_{16}$ after receiving ReqMsg(19,SN), inserts it in Waiting. In Fig. 8(c), process $P_{19}$ after releasing its CS, appends Waiting to token.next and sends the token to process $P_{16}$. In Fig. 8(d) as process $P_{19}$ becomes the explicit token-holding node, after receiving the token and before entering its CS, must inform all nodes in its row by sending InfoMsg(19,Q) in the horizontal path.

When InfoMsg(19,Q) arrived to process $P_{19}$ at the end of its circular path, process $P_{22}$ sends RelMsg(19,true) to process $P_{16}$. Process $P_{19}$ waits until process $P_{16}$ sends Finished(Waiting,i) to it. In Fig. 8(e), process $P_{19}$ after receiving RelMsg(19,true), multicasts RowRel(16,19) to all nodes in its row. In Fig. 8(f), process $P_{16}$ waits until all acknowledge messages from all nodes in its row arrive. Then, in Fig. 8(g) process $P_{16}$ declares that it has finished managing the token by sending Finished(Waiting,i) to process $P_{19}$. Process $P_{19}$ enters its CS after receiving Finished(Waiting,i) in Fig. 8(h).

In time $t_2$, $t_2 > t_1$, processes $P_{15}$, $P_6$, $P_3$ and $P_2$ attempt to invoke their critical sections simultaneously. In Fig. 8(h) these nodes send their ReqMsgs vertically in down direction. ReqMsgs of processes $P_{15}$ and $P_6$ after arriving to informed-nodes, are directly sent to process $P_{19}$ (Fig. 8(i)). Suppose ReqMsgs of processes $P_{15}$ and $P_6$ arrived to process $P_{19}$ respectively, before process $P_{19}$ releasing its CS. Because process $P_6$ has attempted to invoke its CS before arriving ReqMsg(3,SN) to process $P_6$, ReqMsg(3,SN) is inserted in Waiting and will not be transferred to other nodes (PCL-1). Process $P_{19}$ after releasing its CS, has two not responded requests from processes $P_{15}$ and $P_6$ in Waiting. Process $P_{19}$ appends Waiting to token.next. Then process $P_{19}$ extracts existing ReqMsg in head(token.next) which is ReqMsg(15,SN) and directly sends the token to process $P_{15}$ in Fig. 8(j). Process $P_{15}$ after receiving the token, becomes the implicit token-holding process (PCL-2) and enters its CS (Fig. 8(k)). Process $P_{15}$ after releasing its CS, appends Waiting to token.next and passes the token to process $P_6$ in Fig. 8(l). Note that Waiting is empty in this scenario.

In Fig. 8(m) process $P_3$ after receiving the token becomes the implicit token-holding process. Therefore, it appends Waiting to token.next, which is ReqMsg(3,SN) in this scenario. Then it enters its CS. In Fig. 8(n) process $P_6$ after releasing its CS, sends the token to process $P_3$ because ReqMsg(3,SN) is in the head(token.next).

In Fig. 8(o) process $P_3$, before entering its CS, must inform nodes in its row that it is the explicit token-holding process. To inform, process $P_3$ sends InfoMsg(3,Q) to its Right and waits until receives this message again. On the other hand, before sending InfoMsg(3,Q) by process $P_3$, ReqMsg(22,SN), which is sent vertically in down direction by process $P_{22}$, has passed from process $P_2$ in row 1 and column 2. ReqMsg(22,SN) is inserted in P_waiting of all nodes in rows 5, 1, 2 and 3 which are in column 2, i.e. ReqMsg(22,SN) has not arrived to the informed-nodes until now. ReqMsg(22,SN) (which is in P_waiting) through InfoMsg(3,Q) arrives to process $P_3$. Process $P_3$ after appending $Q$ to token.testreq, sends RelMsg(3,false) to process $P_{19}$. Process $P_3$ waits until Finished(Waiting,i) arrives to it from process $P_{19}$.

Then, in Fig. 8(p), process $P_{19}$ multicasts RowRel(19,0) to all nodes in its row. In Fig. 8(q) process $P_{19}$ after receiving Acks from all nodes in its row, sends Finished(Waiting,i) to process $P_{19}$. Process $P_{19}$ by receiving Finished(Waiting,i), appends Waiting to token.next. Then by checking all the existing ReqMsgs in token.testreq one at a time, and comparing $SN$ of each one with the corresponding element in token.seqnum, recognizes all not responded ReqMsgs that are deferred and appends them to token.next. Therefore, ReqMsg(22,SN) is inserted in the rear of token.next. In Fig. 8(r) process $P_{19}$ enters its CS and then releases it. Because ReqMsg(22,SN) is in the head(token.next), process $P_1$ sends the token to process $P_{22}$ in Fig. 8(s). On the other hand, ReqMsg(22,SN), after passing from other nodes in Column 2 arrives to process $P_{22}$ again. Process $P_{22}$ does not send ReqMsg(22,SN) in down direction, the reason being to avoid extra movement of ReqMsg(22,SN) along Column 2. Process $P_{22}$, after receiving the token, continues the algorithm as explained.

With the assistance of this scenario, we tried to describe all aspects of the algorithm.
5 PROOF OF 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

Safety or ME is assured if not more than one node executes its CS simultaneously. For each pair of nodes, one node must release its CS before the other node enters its CS. At first, there is only one token-holding node in our token-based algorithm, this node may remain the token-holding node to continue executing its CS or may release the token and become a non-token-holding node. Because only the token-holding node can enter its CS, it is sufficient to show that just one token-holding node exists in any given time. Only the token-holding node can send the token to just one other node, after which it becomes a non-token-holding node. The token is transferred to the receiver within a limited time. On the other hand, a token cannot be produced and sent by any non-token-holding node.

Theorem 1 (Safety). The algorithm confers safety property.

Proof. We use “reduction to the absurd” for proving safety assurance. Thus we state safety is not assured. As a result, two or more nodes can execute their critical sections, simultaneously. Because in our algorithm only the token-holding node can enter its CS, thus system must be a multi token one. These tokens existed in the system to begin with, or some nodes have produced the tokens or some non-token-holding nodes have sent token messages to other nodes, or the token-holding node can has sent the token to more than one node. Considering the mentioned explanation, these assumptions are impossible. Hence, there is a contradiction. This contradiction then shows that the assumption that more than one node can enter their critical sections simultaneously is incorrect. In the end, safety is assured.

5.2 Liveness Is Assured

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

Theorem 2 (Liveness). The algorithm confers liveness property.

Proof. We prove this by contradiction, too. Therefore, suppose that our 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 other nodes: This assumption is incorrect because in the beginning of the algorithm, P1 is the token-holding process and this token will eventually be sent to another node (based on the assumptions of the algorithm).

2. The token-holding node does not eventually get the information about other nodes’ requests: The requested message of process P1 is either directly arrived to the token-holding node or it is arrived to the node which has attempted to invoke its CS, but has not obtained the token yet (PCL-1), or it is arrived to one of the informed-nodes (the token-holding node is one of the informed-nodes) when ReqMsg(i,SNi) is sent in down direction. In the first case, the token-holding node gets ReqMsg(i,SNi), immediately. In the second case, the previous requester node which has blocked ReqMsg(i,SNi) eventually becomes the implicit token-holding node and then inserts ReqMsg(i,SNi) in token.next. In the last case, through informed-nodes, ReqMsg(i,SNi) is inserted in token.next. There is a special situation in which ReqMsg(i,SNi), without reaching to an informed node, on its moving route is inserted in P_waiting of its column nodes. However, eventually one of the nodes in this column becomes an informed-node. Therefore, ReqMsg(i,SNi), after inserting in token.testreq, if it was not already inserted in token.next, is inserted in this queue. Therefore, the first assumption is incorrect and the token-holding node knows which nodes attempt to invoke their critical sections.

3. The token-holding node keeps the token forever: The token-holding node finishes executing its CS in a limited amount of time. Then, it releases the CS and removes its ReqMsg from either head(token.next) or the head of its Waiting. If the token-holding node is neither interested in entering its CS nor its request appears in head(token.next), appends its Waiting to token.next. If token.next is not empty, the token-holding node extracts existing ReqMsg in head(token.next), which we assume it is ReqMsg(i,SNi), and sends the token to process Pf. If token.next is empty, then the token-holding node waits until receiving a ReqMsg, after that, similar to what is mentioned before, sends the token to process Pf. This contradiction then shows that the anti-liveness assumption cannot be true.

4. Messages do not arrive to the destination node: Based on assumptions in the algorithm, network is error-free and nodes act correctly, thus this statement is incorrect, too.

5. Nodes’ requests for entering their critical sections in token.next are not responded: token.next is a FIFO queue without priority. Therefore, a node which its ReqMsg is inserted in token.next, eventually receives the token. Therefore, this assumption cannot be true, either.

In the end, liveness is assured.

6 PERFORMANCE ANALYSIS

In some definitions, a system is called distributed if message transmission delay is not negligible compared to the time between consecutive events in a single process [1]. Therefore, 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 messages exchanged per request for a CS (Message Complexity) to evaluate the algorithm performance. The performance of a ME algorithm is often studied under two spe-
cial loading conditions, i.e., light load and heavy load. We compare the behavior of our algorithm mainly based on the number of message exchanges with some famous algorithms and summarize the results in Table 1, under these two kinds of loads.

Concerning message exchanges, the worst case behavior of the algorithm coincides with what is mentioned in the Main Idea (Subsection 4.1). This is under the assumption that the requesting process, \( P_r \), and informed-nodes are in the furthest possible distance in the proposed communication network. For example, process \( P_r \) is placed in row one and the informed-nodes are located in row \( \sqrt{N} \). Thus, the request of process \( P_r \) arrives to one of the informed-nodes with \( \sqrt{N} - 1 \) message exchanges, and that informs-node sends \( \text{ReqMsg}(i,SN) \) to the token-holding node, e.g. process \( P_t \), with at the most one message exchange. Therefore, up to this step, \( \sqrt{N} \) message exchanges are required for arriving \( \text{ReqMsg}(i,SN) \) to the process \( P_r \). When process \( P_r \) releases its CS, it sends the token to process \( P_t \) by one message exchange, and process \( P_t \), after receiving the token, must inform all nodes in its row that it is the explicit token-holding node. This requires \( \sqrt{N} \) message exchanges. Then, process \( P_t \) sends \( \text{RelMsg}(i,\text{false}) \) to process \( P_r \) by one message exchange and, upon, receiving process \( P_r \) multicasts \( \text{RowRel}(i,0) \) to all nodes in its row. The latter sends requires \( \sqrt{N} - 1 \) message exchanges and for receiving the \( \text{Acks} \) from all these nodes another \( \sqrt{N} - 1 \) message exchanges are required. Finally, process \( P_t \) sends the \( \text{Finished}(\text{Waiting}) \) to process \( P_r \) by one message exchange and process \( P_r \) after receiving this message, can enter its CS without any further message exchanges. Therefore, in the worst case (very light load conditions), our algorithm requires \( 4\sqrt{N} + 1 \) message exchanges which are fewer than what is needed by similar algorithms (for example [1, 8, 9, 10, 11, 13, 14, 15, and 16]). However, the worst case scenario does not occur frequently.

Suppose \( W \) message exchanges are required for arriving \( \text{ReqMsg}(i,SN) \) to one of the informed-nodes. Sometimes it is possible that \( \text{ReqMsg}(i,SN) \), on its transfer path in down direction, is inserted in the \( \text{Waiting} \) of a node which had already requested to enter its CS (PCL-1); which requires \( \sqrt{N} - 2 \) message exchanges at the most. There are \( V \) message exchanges required for sending \( \text{ReqMsg}(i,SN) \) from the informed-node to process \( P_t \), where process \( P_t \) is the token-holding node, \( 0 \leq W \leq \sqrt{N} - 1 \) and \( V \) equals to zero or one. \( W \) is zero if the requesting node is one of the informed-nodes and \( V \) is zero if the mentioned informed-node is the token-holding node. If the token-holding node is not the token-holding node, one message exchange is required for sending the token to it. The best case occurs when before process \( P_r \) enters its CS some other \( \text{ReqMsgs} \) except \( \text{ReqMsg}(i,SN) \) exist in \text{token.next} (PCL-2). If the requesting node is not the token-holding node, the number of overall message exchanges in this case is between \( 2 \) and \( 2\sqrt{N} + 1 \), which is also better than many similar algorithms.

### 6.1 Performance under Heavy Demand

Consider a heavy load situation in which, in time \( t_1 \), all nodes attempt to invoke their critical sections, simultaneously. Besides, each node after releasing its CS, attempts to immediately invoke its CS again. On the other hand, suppose process \( P_i \) (in row \( x \) and column \( y \)) is the explicit token-holding process. Assume that at first process \( P_i \) is executing its CS. Suppose in time \( t_2 \) \((t_2 > t_1)\), before process \( P_i \) releases its CS, \( \text{ReqMsgs} \) of all nodes in row \( x \) have arrived to process \( P_i \) and \( \text{ReqMsgs} \) of all nodes in row \( r \) \((\forall r, 1 \leq r \leq \sqrt{N} \text{ and } r \neq x) \) arrive to nodes in the next row \((\text{i.e. row } (r \text{mod} \sqrt{N}) + 1)\) on their transfer path in down direction (See Fig. 9(a)). These \( \text{ReqMsgs} \) will not be passed to their next row \((\text{i.e. row } ((r + 1) \text{mod} \sqrt{N}) + 1)\) because all nodes in their next row had attempted to invoke their critical sections in time \( t_1 \) (PCL-1). Up to this step, \((N - 1)\) messages are exchanged. From now on, \( \text{ReqMsgs} \) do not move and only the token will be transferred from one node to another. Process \( P_r \), after releasing its CS, appends \( \text{Waiting} \) (consists of \( \text{ReqMsgs} \) of all nodes in its row) to \text{token.next}, extracts \text{head(token.next)}, which is assumed to be \( \text{ReqMsgs}(f,SN) \), and sends the token to process \( P_t \). It is possible for the token to be transferred to other nodes with respect to \text{head(token.next)} which is shown in Fig. 9(b). Process \( P_r \) after receiving the token, becomes the implicit token-holding process. It, first, appends \( \text{Waiting} \) to \text{token.next} and then enters its CS. Note that, only one \( \text{ReqMsg} \) which belongs to the previous node of process \( P_t \) in the \( P_t \)'s column exists in \( \text{Waiting} \). Process \( P_r \) after releasing its CS, continues the algorithm. Hence, the token transfers between nodes that are not satisfied. Generally, \((N - 1)\) messages for the token movement must be exchanged to satisfy \((N - 1)\) requesting nodes. The last node, say process \( P_r \), which is not satisfied yet and is located in row \((x \text{mod} \sqrt{N}) + 1)\), requires \( 4\sqrt{N} + 1 \) message exchanges to enter its CS because it is the explicit token-holding process (similar to the scenario in Fig. 3). Therefore, a total of \( 2(N - 1) + 4\sqrt{N} + 1 \) messages are exchanged among the \( N \) nodes when the system is under heavy load. As a result, the number of messages per CS entry is:

\[
(2N + 4\sqrt{N} - 1)/N = 2 + 4\sqrt{N}/(N - 1)/N.
\]

So, for the higher number of nodes the number of message exchanges is lower with the minimum of two. If the number of nodes is more than four, then algorithm performs better than Raymond's algorithm, in the heavy demand situations.

### 7 Simulation Results

Because of lack of required facilitates we are unable to setup a reasonable size distributed system. Therefore, we carried out a simulation (similar to [14, 16, 24, and 25]) to evaluate the average message complexity of the info-based algorithm and to compare its performance with several other algorithms ([12, 14, 15]), which are selected from the following categories respectively: hybrid token-based, permission-based, and quorum-based.

The simulation parameters are shown in Table 2, which is similar to [25]. We assumed that local computation and message transmission do not consume local time. The time of first request of every process to enter the CS is selected randomly, while the CS execution time and message delay is assumed to be constant. The number of trials (simulations) is 100 for each combination of simulation parameters, and each simulation terminates when the total number of CS entries by all processes reaches 10000. We define two scenarios (similar to [25]):
• Scenario 1. In this scenario three cases \( (T_{idle} = 10^{-5}, 10^{0}, \text{ and } 10^{5}) \) are assumed. In each case the number of nodes is variable \((N=4, 9, 16, \ldots, 100)\) while non-CS time is fixed.

• Scenario 2. In this scenario, three cases \((N=25, 36, \text{ and } 100)\) are supposed. In each one the number of nodes is fixed while non-CS time is variable \((T_{idle} = 10^{-5}, 10^{0}, \text{ and } 10^{5})\).

Although the proposed algorithm assumes an asynchronous DS, our simulation model is a synchronous one, with a global clock. The simulation was performed via Matlab software. The computing environment is given as follows:

- AMD Athlon™ 64 X2 Dual Core Processor 4200+ (a 2.21-GHz clock) and a 1.00-Gbyte memory,
- Microsoft Windows XP Professional (32 bit) Version 2008, Service Pack 3, and
- Matlab Software: Matlab 7.10 (R2010a).

In the following, we investigate the simulation results using Figs. 10 and 11. Fig. 10 shows the relations between the number of nodes \((N)\) and the average number of messages per CS entry in different loads (Scenario 1). When \(T_{idle}\) is large (small), the system is in light (heavy) demand situations. For case that the load is heavy \((T_{idle} = 10^{5})\; \text{see Fig. 10(a)}\) the message complexity of the proposed algorithm is very low, because the received requests are blocked in the requesting processes (PCL_1) and, generally, most of the processes enter the CS when each of them becomes implicit token-holding process (PCL_2). In info-based algorithm, for light load situations \((T_{idle} = 10^{-5})\; \text{see Fig. 10(b)}\) more messages are exchanged, because every requester must become explicit token-holding node for each entry of the CS. Fig. 10(c) indicates an average load on the DS when \(T_{idle} = 1\). It is clear that info-based algorithm performance is acceptable.

Fig. 11 shows the relations between \(T_{idle}\) and the average number of messages per CS invocation when \(N=25, 64, \text{ and } 100\) (Scenario 2). Also, this figure indicates that the proposed algorithm is scalable on the number of nodes.

As shown in Figs. 10 and 11:

- Multi token algorithm [14] presents a uniform message complexity which is \(N\) messages per CS invocation on both light and heavy demand.
- Maekawa’s algorithm [15] gives a better message complexity in comparison with multi token because it is quorum-based and needs \(c(\sqrt{N} - 1)\) messages \((3 \leq c \leq 5)\). In contrast to info-based, Maekawa’s algorithm has the worst (best) performance on heavy (light) demand conditions. This algorithm performs better than info-based algorithm only in very light demand conditions.
- Hybrid algorithm [12] applies perpetual mobility of the token. This mobility wastes resources of the DS in very light situations because the token sometimes circulates uselessly. Therefore, hybrid performance is low in these conditions. But it is bounded to \(\sqrt{N}\) in the heavy demand conditions.

8 DISCUSSION

8.1 Improvement of the array

The algorithm presented here can be implemented using a logical two dimensional array with \(u\) rows and \(v\) columns. With this structure, a request of process \(P_i\) for executing its CS reaches an informed-node with \(u-1\) message exchanges, in the worst case (similar to what is mentioned in Section 6). It reaches the token-holding node with one message exchange at the most. The total number of messages exchanged, so far, becomes \(u\). Furthermore, there is one message exchange for token movement, \(v\) message exchanges for circulating \(InfoMsg\) in process \(P_i\)'s row, one message exchange for sending \(RelMsg\), \(v-1\) message exchanges for multicasting \(RowRel\), \(v-1\) message exchanges for receiving \(Acks\), and, finally, one message exchange for sending \(Finished\) is required. Therefore, in the worst case (light load conditions), \(u+3v+1\) message exchanges are needed. Similar to what is mentioned in Section 6, this algorithm requires between \(2\) and \(u+1\) message exchanges, in the best case.

To improve the minimum number of message exchanges in light load conditions, we define the dimensions of the two-dimensional array as follows:

Let \(f = u+3v+1\), we have \(N = uv\). Thus, \(f = u + 3N/u + 1\). To find the minimum \(f\) we have to set \(\partial f / \partial u = 0\). But \(\partial f / \partial u = 1 - 3N/u^2\). Therefore, we obtain \(u = \sqrt{3N}\) and \(v = \sqrt{3N} / 3\).

If the number of rows is \(\sqrt{3N}\) and the number of columns is \(\sqrt{3N} / 3\) (Note that the number of rows and columns must be rounded to integer numbers), the number of message exchanges, in the worst case, is approximately equal to \(3.5\sqrt{N}\). Therefore, for the optimal message exchanges, the number of nodes in row and column of the array should be calculated.

8.2 Limitations of the model

There are some limitations in the model which can lead to future research. The limitations are as follows.

- Fault tolerance: Generally, the algorithms that incorporate fault tolerance aspects base their detection mechanisms on the use of timeouts (e.g. [10, 13, and 14]). Some algorithms (e.g. [15]), by assuming that a node failure can be detected by another node and a failed node is removed from the system, consider a simple approach for node removal which is to have another node to take over the role of the failed node. It will cause the overtaking node to play a somewhat greater role. According to our algorithm and the wraparound two-dimensional array logical topology, considering fault tolerance makes the analysis and design more complicated. Therefore we suppose that the network is error-free and the processes operate properly (similar to [1, 4, 9, 11, 24, 25, 26, and 27]). We left fault tolerance aspects for the future work.

- Scalability: If number of nodes is considered constant, in the heavy demand situation the proposed algorithm performs better than many other algorithms as shown in Fig. 11 and also Table 1. Although, in the light demand situation, some al-
The algorithm may perform better than our algorithm. We evaluated the performance under variant number of nodes in Fig. 10. In this case, the performance of the algorithm is acceptable when the number of nodes is increasing. However, in the worst case, performance of the algorithm is $O(\sqrt{N})$. Therefore, we do not claim its complete scalability.

- Ad-hoc connection: We presented an algorithm to solve DME problem for applications such as replicated data management and atomic commitment in distributed databases. This algorithm does not support mobility of nodes or energy consumption considerations. However, it may function in ad hoc networks directly on top of routing protocols but we do not recommend it.

9 Conclusion

The algorithm presented in this paper is an info-based approach to solve the distributed mutual exclusion problem. Our algorithm is based on token, and a process that obtains the token can enter a critical section. The algorithm is fully distributed. On the other hand, we proved that it satisfies the requests of entering the critical sections, correctly.

This algorithm can be generalized to more than one resource. Thus, every node may have separate queues for requests to enter each CS. For every resource there should be a dedicated token and algorithm executes separately per resource. Therefore, using this algorithm, it is possible to execute multiple critical sections for different resources in each process.

Generally, in very light demand conditions, the number of necessary message exchanges, $4\sqrt{N} + 1$, is more than heavy demand conditions, 2, per CS invocation. The performance of the algorithm is much better in comparison with many other algorithms and requires fewer message exchanges, especially in the heavy load situations. Therefore, we recommend that the algorithm should be used on resources with high utilization or implemented in the large distributed systems. In a large DS, it is more probable that some requesting nodes exist at any given time; therefore, the algorithm is not executed in very light demand situations. The algorithm also recommended for the applications that the upper bound of message exchanges is important. The development of an algorithm that adapts to group mutual exclusion problem is left as a future work.

References


Fig. 2
Fig. 3

1. ReqMsg
2. token
3. InfoMsg
4. RelMsg
5. RowRel
6. Ack
7. Finished
/*i, j, f: identifiers of processes P_i, P_j, and P_f respectively.
P_i: process exists in row m and column n which is attempting to enter its CS.
P_j: process exist in row x and column y.
INSERT: entering elements in a queue that have not entered already.
APPEND: removing elements from one queue and then inserting them in another queue.
EXTRACT: using an existing element in a queue without deleting it from that queue.*/

token.idexp←k, token.rowexp←a, token.seqnum[1...N]←0, token.next and token.testreq are empty.
For each node: SN←0, Waiting and P_waiting are empty.
For each node except process P_a: SL_token←0.
For each node except nodes of a’s row: CL_token←0.
For each node in row a except process P_a: CL_token←k.
For process P_a: CL_token←k, SL_token←k.
REQUEST THE CS:

\[ SN_i \leftarrow SN_i + 1; \]
\[ \text{CREATE } ReqMsg(i,SN_i); \]
\[ \text{INSERT } (\text{Waiting},\text{ReqMsg}(i,SN_i)); \quad \text{// Inserts } ReqMsg(i,SN_i) \text{ in the rear of Waiting.} \]
\[ \text{IF } (CL_{\text{token}} \neq 0 \text{ and } CL_{\text{token}} \neq i) \text{ THEN} \]
\[ \text{SEND } ReqMsg(i,SN_i) \text{ to } CL_{\text{token}}; \quad \text{// Send request message to the explicit token-holding node.} \]
\[ \text{ELSE IF } (CL_{\text{token}} = 0) \text{ THEN} \]
\[ \text{INSERT } (P_{\text{waiting}},\text{ReqMsg}(i,SN_i)); \]
\[ \text{SEND } ReqMsg(i,SN_i) \text{ to Down}; \]
\[ \text{ELSE} \]
\[ \text{INSERT } (\text{token.next},\text{ReqMsg}(i,SN_i)); \]
\[ SL_{\text{token}} \leftarrow i; \]
\[ CL_{\text{token}} \leftarrow 0; \]
\[ \text{WAIT } (\text{token}); \quad \text{// Wait for receiving the token.} \]
\[ \text{WAIT } (\text{head(token.next)} = \text{ReqMsg}(i,SN_i)); \quad \text{// This statement avoids executing the CS by the token-holding node continuously while other nodes attempting to invoke their critical sections.} \]
\[ \text{WAIT } (CL_{\text{token}} = i); \quad \text{// Wait until } CL_{\text{token}} = i. \]
\[ SL_{\text{token}} \leftarrow i; \]
RECEIVE A MESSAGE BY PROCESS $P_i$:

SWITCH (message type)

1. **CASE ReqMsg($j,SN_j$):**
   - **CASE ReqMsg($j,SN_j$):**
     - IF ($SL_{token} = i$ or $CL_{token} = i$) THEN INSERT ($Waiting_i$, ReqMsg($j,SN_j$));
     - ELSE IF ($Waiting_i$ is not empty) THEN INSERT ($Waiting_i$, ReqMsg($j,SN_j$));
     - ELSE IF ($CL_{token} \neq 0$) THEN SEND ReqMsg($j,SN_j$) to $CL_{token}$;
     - ELSE IF ($i \neq i$) THEN
       - INSERT ($P_{waiting_i}$, ReqMsg($j,SN_j$));
       - SEND ReqMsg($j,SN_j$) to Down;

2. **CASE token:**
   - APPEND ($token.next$, $Waiting_i$);
   - IF (there is only one element in $token.next$) THEN // $P_i$ becomes the explicit token-holding process.
     - $SL_{token} \leftarrow i$;
     - CREATE $Q$;
     - EMPTY $Q$;
     - APPEND ($Q$, $P_{waiting_i}$);
     - CREATE InfoMsg($i$, $Q$);
     - SEND InfoMsg($i$, $Q$) to Right;
   - ELSE // $P_i$ becomes the implicit token-holding process.
     - $CL_{token} \leftarrow i$;
     - APPEND ($token.testreq$, $Q$);
     - FREE $Q$;
     - IF ($token.rowexp = m$) THEN // $InfoMsg$ after passing a circular path arrives to its creator.
       - $sw \leftarrow true$;
     - ELSE
       - $sw \leftarrow false$;
     - SEND RelMsg($i$, $sw$) to $token.idexp$;

3. **CASE InfoMsg($j$, $Q$):**
   - IF ($j \neq i$) THEN
     - $CL_{token} \leftarrow j$;
     - APPEND ($Q$, $P_{waiting_i}$);
     - SEND InfoMsg($i$, $Q$) to Right;
   - ELSE // $InfoMsg$ after passing a circular path arrives to its creator.
     - APPEND ($token.testreq$, $Q$);
     - FREE $Q$;
     - IF ($token.rowexp = m$) THEN // The new and the old explicit token-holding nodes are in the same row.
       - $sw \leftarrow true$;
     - ELSE
       - $sw \leftarrow false$;
     - SEND RelMsg($i$, $sw$) to $token.idexp$;

4. **CASE RelMsg($j$, $sw$):**
   - NumAck $\leftarrow 0$;
   - IF ($sw = false$) THEN SEND RowRel($i$, 0) to all nodes in Row $m$ except process $P_i$;
   - ELSE SEND RowRel($i$, $j$) to all nodes in Row $m$ except process $P_i$;
   - WHILE ($NumAck < \sqrt{N} - 1$); // Process $P_i$ waits until receives all Acks.
     - IF ($sw = false$) THEN $CL_{token} \leftarrow 0$;
     - ELSE $CL_{token} \leftarrow j$;
     - $SL_{token} \leftarrow 0$;
     - SEND Finished($Waiting_i$) to $j$;

5. **CASE RowRel($i$, $j$):**
   - IF ($i \neq l$) $CL_{token} \leftarrow l$;
   - SEND Ack($i$) to $j$;

6. **CASE Ack($j$):**
   - NumAck $\leftarrow NumAck + 1$;

7. **CASE Finished($Waiting_i$):**
   - APPEND ($token.next$, $Waiting_i$);
   - WHILE ($token.testreq$ is not empty)
     - EXTRACT head($token.testreq$) which is ReqMsg($f,SN_f$);
     - IF ($token.seqnum[f] < SN_f$) THEN
       - $token.seqnum[f] \leftarrow token.seqnum[f] + 1$; // $token.seqnum[f] \leftarrow SN_f$.
     - INSERT ($token.next$, ReqMsg($f,SN_f$));
     - REMOVE ReqMsg($f,SN_f$) from head($token.testreq$);
RELEASE THE CS:

\[
\begin{align*}
&CL_{\text{token}} \leftarrow 0; \\
&\text{IF } (\text{ReqMsg}(i,SN_i) \text{ exists in } \text{head}(\text{token.next}) \text{ or } \text{head}(\text{Waiting},i)) \text{ THEN REMOVE it;} \\
&\text{token.seqnum}[i] \leftarrow SN_i; \\
&\text{IF } (\text{token.idexp} = i) \text{ THEN} \\
&\quad \text{WHILE } (\text{Waiting},i \text{ and } \text{token.next} \text{ are empty}); \\
&\text{ELSE} \\
&\quad SL_{\text{token}} \leftarrow 0; \\
&\quad \text{APPEND } (\text{token.next},\text{Waiting},i); \quad \text{//Removing all elements existing in } \text{Waiting},i \text{ and inserting elements in } \text{token.next}. \\
&\text{IF } (\text{head}(\text{token.next}) \text{ is } \text{ReqMsg}(i,SN_i)) \text{ THEN } CL_{\text{token}} \leftarrow i; \\
&\text{ELSE} \\
&\quad \text{EXTRACT head}(\text{token.next}) \text{ which is } \text{ReqMsg}(f,SN_f); \\
&\quad \text{SEND token to } f;
\end{align*}
\]
Fig. 8
Fig. 9(a)

- The informed-node
- The requesting node
- Sending a message
- ReqMsg
- Explicit token-holding node
Fig. 9(b)

- The informed-node
- The requesting node
- Sending the token
- Probably sending the token
- The old explicit token-holding node
- The new implicit token-holding node
Plot of the number of message exchanges per cs invocation

- Multi-token
- Info-based
- Maekawa
- Hybrid
Plot of the number of message exchanges per cs invocation.

- Multi-token
- Info-based
- Maekawa
- Hybrid

Nodes

#message/cs

Fig. 10(b)
Plot of the number of message exchanges per cs invocation

- Multi-token
- Info-based
- Maekawa
- Hybrid

#message/cs vs $T_{idle}$
Figure 11(b)

Plot of the number of message exchanges per cs invocation

- Multi-token
- Info-based
- Maekawa
- Hybrid

#message/cs vs. $T_{idle}$
TABLE 1
Evaluating of presented algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Network Logical Topology</th>
<th>Requires Fully Connected Topology</th>
<th>Equal order of sending and receiving messages (^2)</th>
<th>Equal order of constructing and satisfying CS requests (^3)</th>
<th>Message complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Worst case</td>
</tr>
<tr>
<td>Token-ring [8]</td>
<td>Logical Structure-Based</td>
<td>Perpetual mobility of the token</td>
<td>x</td>
<td>x</td>
<td>N</td>
</tr>
<tr>
<td>Suzuki-Kasami [9]</td>
<td>Broadcast-Based</td>
<td>Token asking</td>
<td>✓</td>
<td>x</td>
<td>N</td>
</tr>
<tr>
<td>Raymond [10]</td>
<td>Logical Structure-Based</td>
<td>Token asking</td>
<td>✓</td>
<td>x</td>
<td>2((N-1))</td>
</tr>
<tr>
<td>Naimi-Trehel [11]</td>
<td>Logical Structure-Based</td>
<td>Token asking</td>
<td>✓</td>
<td>x</td>
<td>N</td>
</tr>
<tr>
<td>Info-based Algorithm</td>
<td>Logical Structure-Based</td>
<td>Info-based WTDA(^5)</td>
<td>✓</td>
<td>✓</td>
<td>4(\sqrt{N} + 1)</td>
</tr>
<tr>
<td>Lamport [1]</td>
<td>Broadcast-Based</td>
<td>Non-token-based</td>
<td>✓</td>
<td>✓</td>
<td>3((N-1))</td>
</tr>
<tr>
<td>Ricart-Agrawala [13]</td>
<td>Broadcast-Based</td>
<td>Permission-based</td>
<td>✓</td>
<td>x</td>
<td>2((N-1))</td>
</tr>
<tr>
<td>Multi token [14]</td>
<td>Logical Structure-Based</td>
<td>Permission-based</td>
<td>x</td>
<td>x</td>
<td>N</td>
</tr>
<tr>
<td>Maekawa [15]</td>
<td>Logical Structure-Based</td>
<td>Quorum-based</td>
<td>x</td>
<td>x</td>
<td>5((\sqrt{N} - 1))</td>
</tr>
<tr>
<td>Agrawal-Abbadi [16]</td>
<td>Logical Structure-Based</td>
<td>Quorum-based</td>
<td>✓</td>
<td>✓</td>
<td>((N+1)/2)</td>
</tr>
<tr>
<td>Paydar et al. [4]</td>
<td>Logical Structure-Based</td>
<td>Hybrid</td>
<td>WTDA(^5)</td>
<td>x</td>
<td>4(\sqrt{N})</td>
</tr>
<tr>
<td>Hybrid [12]</td>
<td>Logical Structure-Based</td>
<td>Hybrid</td>
<td>WTDA(^5)</td>
<td>x</td>
<td>2(\sqrt{N})</td>
</tr>
</tbody>
</table>

\(^1\) In logical structure-based algorithms, the sites in the system are assumed to be arranged in a logical configuration like tree, ring, etc., and messages are passed from one site to another along the edges of the logical structure imposed. In the case of broadcast-based algorithms, no such structure is assumed and the requesting site sends messages to other sites in parallel, i.e., the message is broadcast [2].

\(^2\) Algorithm requires that for any two processes \(P_i\) and \(P_j\), the sending messages from process \(P_i\) to process \(P_j\) are received in the same order as they are sent.

\(^3\) In the algorithm CS entering requests are satisfied in the order of their construct.

\(^4\) Without considering the situation in which the requesting node for entering the CS holds the token in token-based algorithms.

\(^5\) Wraparound two-dimensional array.
The table below shows the simulation parameters used:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>the number of processes: 4, 9, 16, …, 100.</td>
</tr>
<tr>
<td>$D_{msg}$</td>
<td>message delay: 0.01 (fixed).</td>
</tr>
<tr>
<td>$T_{cs}$</td>
<td>duration in CS: 0.1 (fixed).</td>
</tr>
<tr>
<td>$T_{idle}$</td>
<td>duration in non-CS: $10^{-5}$, $10^{-4}$, …, $10^{-5}$ (expected value under exponential distribution).</td>
</tr>
</tbody>
</table>
P. Neamatollahi received his B.S. and M.S. degree in computer engineering, with concentration in Parallel Computing and Mutual Exclusion in Distributed Systems respectively, from Islamic Azad University, Mashhad branch, Iran. He has published some conference and journal papers. His research interests are in parallel and distributed computing, wireless sensor networks and fuzzy logic control.

H. Taheri received her B.S. and M.S. degree in computer engineering, with concentration in Dasher Software and Clustering Wireless Sensor Networks respectively, from Islamic Azad University, Mashhad branch, Iran. She has published some conference and journal papers. Her research interests are in parallel and distributed computing, wireless sensor networks and fuzzy logic control.

M. Naghibzadeh received his M.S. in computer science and Ph.D. in electrical engineering (Computer) from University of Southern California (USC). He is currently a full professor at the department of computer engineering, Ferdowsi university of Mashhad, Iran. He has published numerous conference and journal papers and many books. His current research interests include real-time process scheduling, grid computing, knowledge engineering, and semantic web.
Figure legends

Fig. 1. A taxonomy of DME algorithms.

Fig. 2. A proposed logical topology of 25 nodes.

Fig. 3. Orders and types of messages exchanged between the token-holding process, $P_k$, and the requesting process, $P_i$.

Fig. 4. Initialization of the algorithm.

Fig. 5. Pseudo-code of REQUEST THE CS procedure of the algorithm in process $P_i$.

Fig. 6. Pseudo-code of RECEIVE A MESSAGE procedure of the algorithm in process $P_i$.

Fig. 7. Pseudo-code of RELEASE THE CS procedure of the algorithm in process $P_i$.

Fig. 8. The Scenario.

Fig. 9. A heavy load situation: (a) All processes request to enter the CS, (b) The explicit token-holding process, $P_j$, passes the token to a requesting process.

Fig. 10. Scenario 1: (a) $T_{idle}=10^{-5}$, (b) $T_{idle}=10^{-5}$, and (c) $T_{idle}=10^0$.

Fig. 11. Scenario 2: (a) $N=25$, (b) $N=64$, and (c) $N=100$. 