A simple token-based algorithm for the mutual exclusion problem in distributed systems

Peyman Neamatollahi[1] · **Yasser Sedaghat**[1] · **Mahmoud Naghibzadeh**[1]

Published online: 22 February 2017

Abstract Solving the problem of mutually exclusive access to a critical resource is a major challenge in distributed systems. In some solutions, there is a unique token in the whole system which acts as a privilege to access a critical resource. Practical and easily implemented, the token-ring algorithm is one of the most popular token-based mutual exclusion algorithms known in this field's literature. However, it suffers from low scalability and a high average waiting time for resource seekers. The present paper proposes a new algorithm which employs a two-dimensional torus logical structure of N processes and the token-ring algorithm concept. It performs in a way that increasingly raises scalability and reduces the average waiting time of the token-ring algorithm. The token makes a circular movement along the columns of the two-dimensional torus (vertical ring), while the requests for the critical resource make a circular movement along the rows of the torus (horizontal ring). In this algorithm, the number of messages exchanged is between $2\sqrt{N} + 1$ and $3\sqrt{N} + 1$ under light load situations and, under heavy load situations, is at the most three messages per critical section invocation. Thus, in contrast with the leading algorithms, the proposed algorithm has gained significant improvements, in addition to having been proved to operate correctly.

Keywords Mutual exclusion · Critical section · Distributed systems · Token-based algorithm · Token-ring

✉ Peyman Neamatollahi
neamatollahi@mail.um.ac.ir

[1] Department of Computer Engineering, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran

1 Introduction

A distributed system consists of distinct processes which communicate with each other through message passing. In other words, a system is called distributed if the message transmission delay is not negligible compared to the time between the events in a single process [1]. One of the most important aims in distributed systems is to provide an environment conducive to sharing resources [2–11]. Hence, it is possible that several processes simultaneously request a shared resource. On the other hand, atomic actions play a significant role in a distributed environment. They may be applied as mechanisms to provide process synchronization [12]. These actions are used to ensure that inconsistencies do not arise when concurrent activities operate on shared objects.

Every process has a code section, namely the Critical Section (CS), in which a process can access a shared resource. There are many situations in single systems, cloud providers, operating systems, distributed databases, and distributed shared memory, among others, in which a resource must be accessed by only one process at any given time [13–20]. When a process tries to invoke a shared resource, it must first enter its CS. In this way, the process can enjoy exclusive access to the shared resource and prevent other processes from interfering in its work [21].

Algorithms presented to assure mutual exclusion in distributed systems are named as Distributed Mutual Exclusion (DME) algorithms. It is necessary to solve the mutual exclusion problem in order to prevent race conditions and also erroneous results from correct programs [22]. In distributed systems, every node has a partial or incomplete view of the system. Moreover, there is not any shared semaphore or infrastructure in which DME algorithms may be implemented. Consequently, the DME problem should be solved using message exchange.

1.1 Background

The issue of DME, which had been introduced by Dijkstra [23], has been fairly well studied [4,15,17,24–30]. In some solutions, a token is a unique entity which allows a node to enter its CS out of all other nodes that are also attempting to invoke their CSs. Such solutions are addressed as token-based algorithms. Token-based algorithms often apply two strategies to assure mutual exclusion, namely the token-asking method and the perpetual movement of the token [31].

In the token-asking method [32–36], when a node attempts to invoke its CS, it sends a request to other nodes if it does not hold the token and then waits for the token. After receiving the token, the requester node executes its CS and then passes the token to a selected requester node. If there is no requester node in the system, the process holds the token and does not forward it. The present study took advantage of the concept of this method along the rows of the logical two-dimensional torus (described in the next section).

In the perpetual token movement strategy, the token passes through the nodes to allow each of them to enter their CSs one after another. It is, therefore, possible that a node receives the token while it passes over, without ever having sent any request.

Consequently, much communication and processing overhead are imposed on the system, especially in the light load situation in which a few number of processes attempt to simultaneously invoke their CSs. However, for heavy load situations, these kinds of algorithms are highly efficient. Token-ring algorithm [37] belongs to the class of token-based algorithms which employs the perpetual token movement strategy. In this type of algorithm, in order not to forget a process's request, the processes are placed on a unidirectional logical ring, around which the token rotates clockwise or counter-clockwise. If a node receives the token without having requested CS entry, it forwards the token to the process in its proximity on the ring. The token's unidirectional rotation guarantees the liveness property. The major problem with the token-ring algorithm is that it is not scalable; by increasing the number of processes, the average waiting time, before entering its CS, lengthens for the requester node. The current study's algorithm utilized the idea of the token-ring algorithm in the columns of the logical two-dimensional torus. Although there are many DME algorithms in the literature [1, 22,31–33,37], they are not efficient with respect to the number of messages exchanged in all cases while the current work performs well in all cases.

1.2 Contribution

In the current paper, an algorithm for solving the DME problem is proposed for the purposes of decreasing communication overhead and increasing the scalability property in a system having at least one unresponded CS entry request at any given time. The main contribution of this work is to decrease the number of messages required for the heavy load condition by using a two-dimensional torus logical structure. Simultaneously, it limits the number of exchanged messages under light load situations in the worst case. This proposed algorithm is considered as token-based. It uses a two-dimensional torus logical topology to decrease the number of message exchanges. The token moves circularly along the columns of the two-dimensional torus to grant the CS entrance privilege to the nodes while processing CS entry requests by rotating horizontally along the rows of that torus, this is so as to alert all the nodes in the row with a pending request. The role of the joint node between the row consisting of the requester node and the column comprised of the token is to change the direction of the token from a vertical movement to a horizontal one. In other words, when the token reaches a given row, it acquires information on the pending requests for that row and serves those requests. The current study's algorithm assures the safety and liveness properties. It also outperforms better than many other algorithms such as [1,31–34] because it decreases the number of messages exchanged under light (between $2\sqrt{N}+1$ and $3\sqrt{N}+1$), medium (between 3 and $3\sqrt{N}$), and heavy demand (at most 3). It should be mentioned that the present work considers the status in which only one, $\sqrt{N}$, and N node attempt to simultaneously invoke their CSs in light, medium, and heavy load situations, respectively.

1.3 Organization

The rest of the paper is organized as follows: Sect. 2 defines the system model of the algorithm; Sect. 3 provides an informal description of the algorithm, data structures and messages, algorithm details (in which the pseudo-code of the algorithm is shown), and finally, describes the details of the algorithm via a scenario; Sect. 4 proves the correctness of the algorithm. In Sect. 5, the performance of the algorithm is analyzed and then a conclusion is drawn.

2 Model

The present work implements its proposed algorithm on a distributed system consisting of N nodes with no shared memory. Therefore, these nodes communicate by asynchronous message exchange. The communication network is presumed to be error-free. Without generality loss, it is supposed that there is only one process in each node. Hence, the process and node can be used interchangeably.

The message propagation delay is unpredictable but finite, which means that every message will eventually be received. Messages may be received in a different sequence than the sequence in which they were sent. A unique identification number is devoted to each process which is between 1 to N.

In the case of broadcast-based algorithms [1,33,34], no structure is assumed, and the requester sends messages to others in parallel, thus broadcasting the message. In contrast, in logical structure-based algorithms [22,31,32,36–40], the sites in the system are considered to be arranged in a logical configuration, such as a tree or ring, and messages are transferred from one site to another along the edges of the imposed logical structure [2]. The present paper employs a logical structure in the form of a two-dimensional torus. Token movement, which is essential in this algorithm, is based on this logical structure of N nodes. To be exact, each node belongs to two logical rings; the token meets the nodes one by one in the vertical circle, while in the horizontal ring, the token meets them on demand. Also, the CS entry requests are only propagated over the horizontal ring. This is the reason why the current work organizes the logical structure in the form of a two-dimensional torus. It is assumed that $N = d^2$, where d is an integer number and N is the number of processes. Therefore, the proposed algorithm's logical torus has $\sqrt{N}$ rows and $\sqrt{N}$ columns. Each process knows only its neighbors to the right and down, and consequently, it can forward messages to either of them, when needed.

It is supposed that processes perform correctly. A process can access a critical resource only when it is executing its CS and this time is limited. When a process requests to enter its CS, it cannot generate another request before the first one is processed and it is given this chance to use the resource and let it go. It is presumed that there is at least one unresponded request in the system at any given time. Also, CS entry requests may be satisfied out of the order of their creation, such as in the algorithms proposed in [31,33,34,36,37,39,41].

3 Proposed algorithm

This section describes a token-based mutual exclusion algorithm for a logical structure shaped as two-dimensional torus. As long as all nodes are idle, the token starts in any node i and visits each node in its-cyclic-column. Returning to node i, the token moves to the right column and visits all its nodes. After covering all columns of the network, the token returns to node i. Additionally, any node j may spontaneously turn from idle to waiting. In this case, j solicits a sequence of messages, alerting the nodes in its-cyclic-row. Of course, the token eventually meets an alerted node. In this case, the token follows the corresponding line until it meets j. If more than one node moves from idle to waiting, then some light bookkeeping is required to serve all such nodes.

The description of the current study's proposed algorithm is divided into four parts: First, an informal description is illustrated; second, data structures and messages are explained; third, the overall algorithm is presented, and then, in the last part, a scenario is applied to illustrate all aspects of the algorithm in detail.

3.1 An informal description of the algorithm

At first, P_k (which is in a row a and column b of the logical topology) is the token-holder process, and the token circulates downward in column b. For simplification, one can assume that there is only one requester process, say process P_i in row m and column n(m is different from a), which is attempting to invoke its CS. The given location of these two nodes and the messages exchanged between the system nodes in the following scenario are shown in Fig. 1a. Process P_i sends its request to enter the CS along the horizontal ring of its row (in the right direction), and it is forwarded by other processes in row m. Therefore, after receiving process P_i's request all existing nodes in row m are aware that process P_i is waiting to receive the token. As shown in Fig. 1b, when the token arrives at process P_j (which is the common node of the token movement column and the CS requester node's row), process P_j checks its request queue. If it is empty, the token continues its downward movement along the vertical ring. If process P_j itself has previously requested access to the CS, then it can enter its CS, or if there are some other CS entry requests in process P_j's request queue, it sends the token to its neighbor on the right in order to start the token's circular movement on the horizontal ring. With this action, all pending requests in this row are responded to, and the token eventually comes back to process P_j. In this case, the token resumes its vertical movement from process P_j.

To decrease the number of message exchanges, three interesting principles are applied to the algorithm which has a remarkable influence on the overall performance:

- PCL_1: When a requester process, say process P_x, receives a request message (e.g., process P_y's request), it blocks that request because process P_x has already alerted other nodes in its row of the existence of some requests in the same row, therefore, not requiring process P_y's request message to complete its horizontal circulation. In addition to decreasing the number of message exchanges, this principle permits each node to have a local queue of the maximum size of two in order to be able to save an arriving request and its own request if any.
- PCL_2: Regarding PCL_1, if a process is attempting to invoke its CS while holding a pending request in its queue, it blocks its own request and does not send it to the neighbor on the right.
- PCL_3: Circulating the token in a particular column imposes an extra load on the nodes in that column. To prevent this additional burden, it is necessary to change

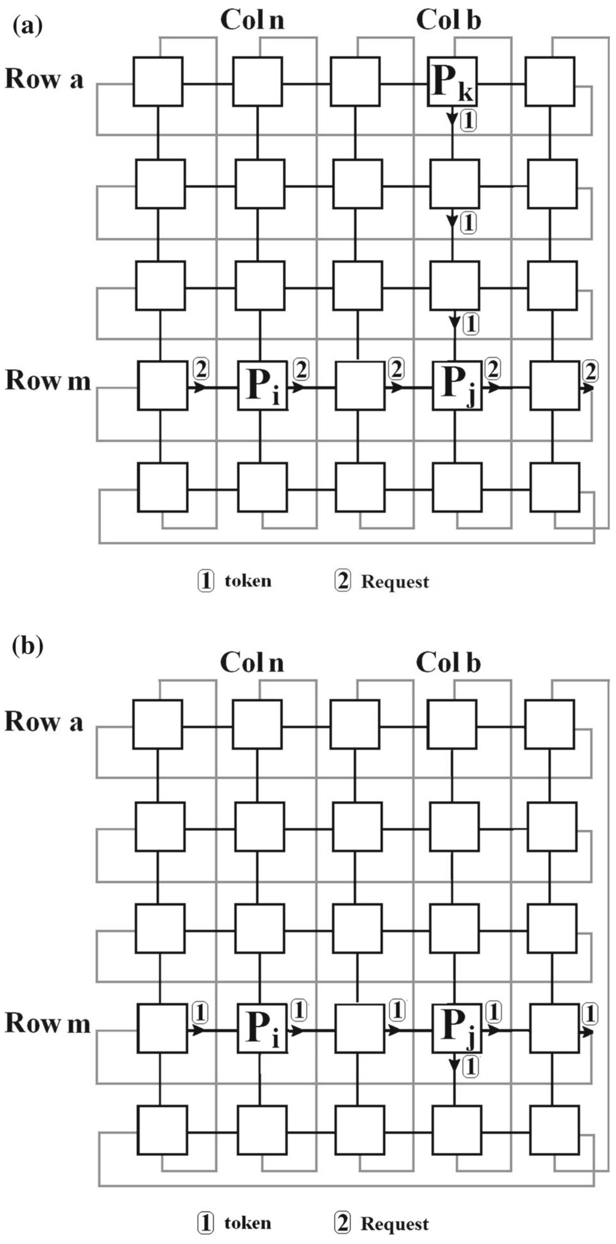

Fig. 1 Total messages exchanged between processes in the algorithm

the token movement column. The principle causes all processes to tolerate an approximately equal workload. The manner in which this principle is implemented is explained in Subsection 3.3.

3.2 Data structures and messages

Data structures and messages used in the proposed algorithm are described as follows. *Request* is a kind of message sent by a process needing to enter its CS, such as P_i. This message contains the identification number of its creator and is denoted as $Request_i$.

Token is a record which is embedded within a message. It contains a field named *RowCounter*. To implement PCL_3, the number of rows passed from where the *Token* started its vertical movement is counted using the *RowCounter* field of the *Token*. Regarding this field, whenever the *Token* completely circulates a column, another field of the *Token*, namely *ChgCol*, is set to TRUE. In other words, the *ChgCol* is a Boolean variable that indicates whether changing columns is necessary or not. If it is TRUE, the *Token* should jump one step to the right to restart its vertical movement in a new column. The other field of the *Token* is *ColCounter* which may have different values in various situations; it is negative when the *Token* is changing its movement column, zero when the *Token* is received from the neighbor above, and positive when the *Token* counts the number of columns while it is navigating a row.

Each node has a local queue of the maximum size of two in order to save the requests named $Waiting_i$. The small size of this queue prevents an overhead due to the handling of high-volume data. Also, each node has a local Boolean parameter called CS-permission, which determines whether the process can enter its CS or not.

In the following algorithm details, it is assumed that process P_k in row a and column b is the token-holder process. Refer to the initialization part of the algorithm in Fig. 2.

3.3 Algorithm details

The behavior of the present study's algorithm is investigated in three situations: (1) process P_i attempts entering its CS, (2) process P_i receives a message from process P_j, and (3) process P_i relinquishes its CS.

Requesting the CS: Process P_i creates $Request_i$ to enter its CS and places it in the rear of $Waiting_i$. If there is only $Request_i$ in this queue, process P_i sends its request to its right node on the horizontal ring. Otherwise, by considering PCL_2, process P_i avoids sending $Request_i$. Then, process P_i waits until it receives the *Token*. Whenever process P_i receives the *Token*, it sets $CS\text{-}permission_i$ to *TRUE* and executes its CS. This action must be performed atomically. In fact, it can be considered as an atomic action. Refer to Lines 1–7 in Fig. 2.

Receiving a Message: When process P_i receives a message from process P_j, it may be in one of two states:

- Process P_i receives $Request_j$. If $Waiting_i$ is empty and $Request_j$ has not completely rotated through the horizontal ring, process P_i inserts that request in its *Waiting* and forwards it to its neighbor on the right. Otherwise, process P_i ignores that request. Refer to Lines 16–23 in Fig. 2.
- The received message is *Token*. In this case, one of the three states may occur:
  1. *Token* is received from the neighbor above of process P_i. As *Token* enters a new row, the *RowCounter* field of *Token* is added by the quantity of one. Now,

Fig. 2 Pseudo-code of the algorithm in process P_i

```
Initialization:
   Token.RowCounter ← 1, Token.ColCounter ← 0,
   Token.ChCol ← FALSE.
   For all processes: CS-permission ← FALSE, Waiting is empty.

Distributed Mutual Exclusion Solver:
1.  CASE REQUESTING THE CS:
2.       CREATE $Request_i$;
3.       INSERT ($Waiting_i$, $Request_i$);
4.         /*inserts $Request_i$ in the rear of $Waiting_i$.*/
5.       IF (there is only $Request_i$ in $Waiting_i$) THEN
6.            SEND $Request_i$ to its right neighbor;
7.       WAIT ($CS\text{-}permission_i$=TRUE);
8.  CASE RELEASING THE CS:
9.       $CS\text{-}permission_i$ ← FALSE;
10.        CLEAR $Waiting_i$;
11.        IF (Token.ColCounter = $\sqrt{N}$) THEN
12.             Token.ColCounter ← 0;
13.             SEND Token to its down neighbor;
14.        ELSE
15.             SEND Token to its right neighbor;
16.    CASE RECEIVING A MESSAGE BY PROCESS $P_i$:
17.       SWITCH (message type)
18.         CASE $Request_j$:
19.           IF ($i \neq j$ AND $Waiting_i$ is empty)
20.                INSERT ($Waiting_i$,$Request_j$);
21.                SEND $Request_j$ to its right neighbor;
22.           ELSE
23.                IGNORE $Request_j$;
24.         CASE Token:
25.           IF (Token.ColCounter = 0) THEN /*State in
26.           which token is received from the upper neighbor.*/
27.                Token.RowCounter ← Token.RowCounter +1;
28.                IF (Token.RowCounter = $\sqrt{N}$) THEN
29.                     Token.ChCol ← TRUE;
30.                     Token.ColCounter ← -1;
31.                     SEND Token to its right neighbor;
32.                ELSE IF ($Waiting_i$ is empty) THEN
33.                     /*There is not any request in this row.*/
34.                     SEND Token to its down neighbor;
35.                ELSE
36.                     Token.ColCounter ← 1;
37.                     IF (There is $Request_i$ in $Waiting_i$) THEN
38.                          $CS\text{-}permission_i$ ← TRUE;
39.                     ELSE
40.                          CLEAR $Waiting_i$;
41.                          SEND Token to its right neighbor;
42.           ELSE IF (Token.ColCounter > 0) THEN /*State in
43.           which token is received from the left neighbor
44.           while it has not completed its horizontal circulation.*/
45.                Token.ColCounter ← Token.ColCounter +1;
46.                IF (There is $Request_i$ in $Waiting_i$) THEN
47.                     $CS\text{-}permission_i$ ← TRUE;
48.                ELSE
49.                     CLEAR $Waiting_i$;
50.                     IF (Token.ColCounter = $\sqrt{N}$) THEN
51.                          Token.ColCounter ← 0;
52.                          SEND Token to its down neighbor;
53.                     ELSE
54.                          SEND Token to its right neighbor;
55.           ELSE IF (Token.ColCounter = −1) THEN
56.           /*State in which token is changing its column.*/
57.                Token.ChCol ← FALSE;
58.                Token.RowCounter ← 1;
59.                IF ($Waiting_i$ is empty) THEN
60.                     Token.ColCounter ← 0;
61.                     SEND Token to its down neighbor;
62.                ELSE
63.                     Token.ColCounter ← 1;
64.                     IF (There is $Request_i$ in $Waiting_i$) THEN
65.                          $CS\text{-}permission_i$ ← TRUE;
66.                     ELSE
67.                          CLEAR $Waiting_i$;
68.                          SEND Token to its right neighbor;
```

by checking this field, it is recognizable whether *Token* has completely rotated a vertical ring or not. If so, it should continue its vertical movement in the right column of process P_i. In the case that the vertical rotation of *Token* has not yet been completed and if there is not any request in the $Waiting_i$ queue, *Token* is sent to the node below process P_i. Otherwise, if $Request_i$ is not in $Waiting_i$, then process P_i clears $Waiting_i$ and sends *Token* to its neighbor on the right. In contrast, if $Request_i$ is in $Waiting_i$, then process P_i can enter its CS. Refer to Lines 24–41 in Fig. 2.

2. *Token* is received from the neighbor on the left of process P_i. In this case, process P_i can enter its CS if this has been previously requested. If not, process P_i clears $Waiting_i$ and sends *Token* to its neighbor on the right if *Token* has not completed a rotational movement on this row. Otherwise, process P_i sends *Token* to the node below to resume its vertical movement. Refer to Lines 42–54 in Fig. 2.
3. *Token* is received from the neighbor on the left of process P_i in order to change the vertical movement column of *Token*. Process P_i checks its *Waiting* and, as mentioned in Case 2, makes a decision whether to send *Token* on its regular course, that is, to start its horizontal circulation with the aim of responding to requests from nodes located in this row or to send *Token* to the node below in order to start its vertical movement in this new column; this action is performed when $Waiting_i$ is empty. Refer to Lines 55–68 in Fig. 2.

Releasing the CS: One can now suppose a situation in which process P_i finishes its CS execution. First, it makes $CS\text{-}permission_i$ equal to *false* so that it cannot immediately enter its CS, so as to prevent other needy processes from starving. Then, process P_i clears $Waiting_i$. If *Token* has completed its horizontal movement, process P_i sends *Token* to its neighbor process below; otherwise, process P_i sends *Token* to its neighbor on the right. Refer to Lines 8–15 in Fig. 2.

3.4 A scenario

In addition to describing the pseudo-code of the proposed algorithm's formal presentation, the present study utilizes a scenario to explain actions in detail and with the assistance of Fig. 3. In Sect. 3, it is assumed that P_k is the token-holder process. As shown in Fig. 3a, process P_8 (here k is equal to 8) sends *Token* in a downward direction, while process P_{16} attempts to invoke its CS. Therefore, process P_{16} inserts $Request_{16}$ in $Waiting_{16}$ and then sends $Request_{16}$ to its neighbor on the right so that all nodes in its row are eventually informed of its need to enter CS.

Now, process P_{18}, after receiving *Token*, checks its *Waiting*. $Waiting_{18}$ is empty because $Request_{16}$ has not yet arrived to process P_{18}. Thus, process P_{18} sends *Token* in a downward direction to continue the vertical movement of *Token* in column 3 (see Fig. 3b).

On the other hand, process P_{23}, after obtaining *Token*, passes *Token* to the next node in the vertical movement of *Token*. This action is continued through process P_3 in Fig. 3c. Also, the request of process P_{16} has arrived at all nodes in row 4 and is inserted in the *Waiting* queues of those nodes.

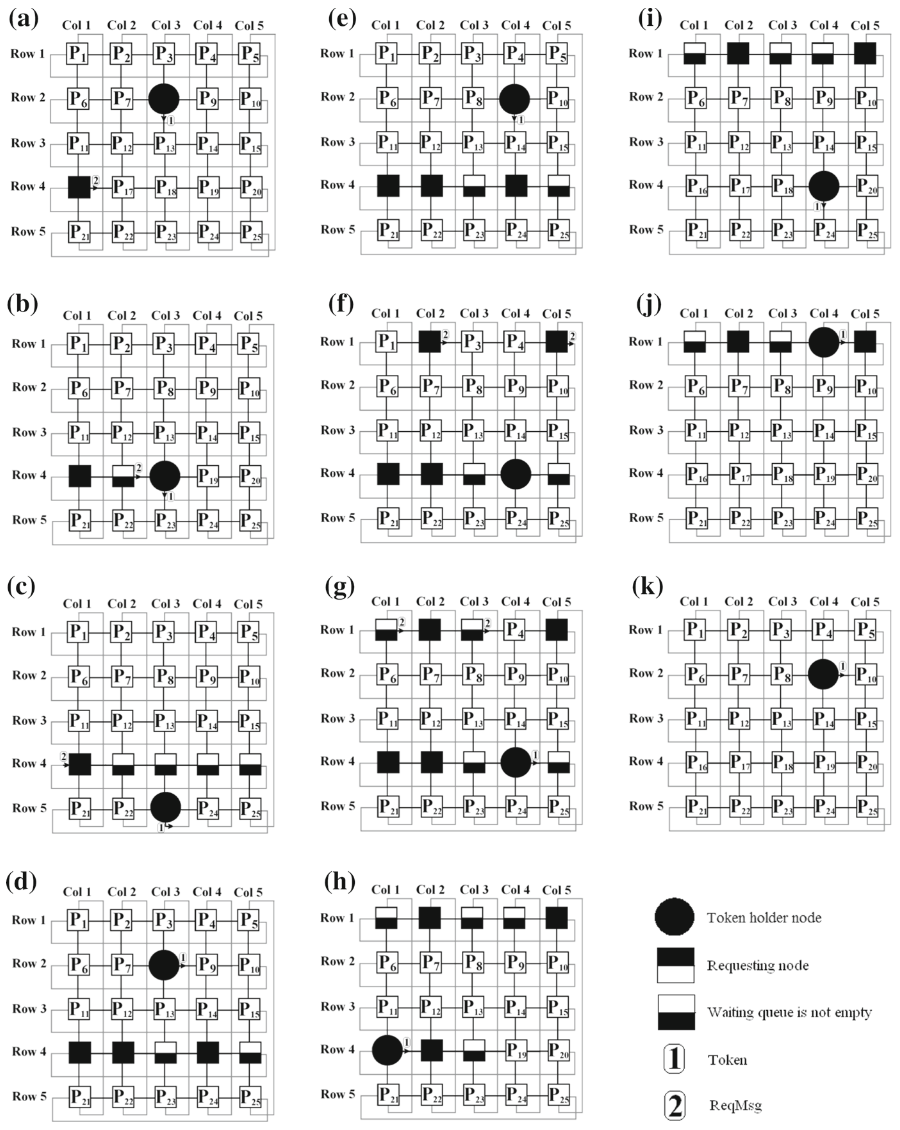

Fig. 3 A scenario

In Fig. 3d, processes P_{17} and P_{19}, which are attempting to invoke their CSs, find their *Waiting*s non-empty. Therefore, they just insert their *Requests* in corresponding *Waiting*s and do not forward those messages to their next node according to PCL_2. This action leads to fewer message exchanges. On the other hand, process P_8, after receiving *Token*, passes it to the node on the right. This changing of columns is due to the finalizing of the circular movement of *Token* in column 3.

In Fig. 3e, the circular movement of *Token* begins in column 4 through process P_9. This action continues until *Token* arrives at process P_{19}.

After receiving *Token*, process P_{19}, in Fig. 3f, enters the CS and then clears $Waiting_{19}$. Processes P_2 and P_5 simultaneously request to enter their CSs.

After executing its CS, process P_{19} sends *Token* to process P_{20}, as shown in Fig. 3g. Because process P_{20} has not made any request itself, it clears its queue and forwards *Token* to its neighbor on the right (process P_{16}).

As depicted in Fig. 3h, when receiving *Token*, process P_{16} enters its CS. Process P_{16} then clears its *Waiting* after releasing its CS and then sends *Token* to process P_{17}. Upon receiving *Token*, process P_{17} enters its CS, releases it and then passes *Token* to the next process. After receiving *Token*, process P_{18} clears its queue and forwards *Token* to process P_{19}. On the other hand, $Request_5$ is inserted in the *Waiting* of processes P_5 and P_1. Similarly, $Request_2$ is placed into the *Waiting* of processes P_2, P_3, and P_4. It is notable that these two requests will not be forwarded to other nodes according to PCL_1.

In Fig. 3i, as a horizontal ring is completely traversed by *Token*, process P_{19} sends it to the neighbor blow so that *Token* can resume its vertical movement.

Token keeps its vertical movement over the nodes until it reaches process P_4 whose *Waiting* is non-empty (Fig. 3j). Now, *Token* is responsible for responding to all pending requests in row 1. Besides, all the nodes in this row clear their *Waiting*s while *Token* passes over them. As *Token* completes the horizontal rotation, it moves down and comes back to the first location where it started the vertical circulation (process P_9).

Now, the *Token* must change its column on its regular course (Fig. 3k). Process P_{10} continues the algorithm as explained.

With the assistance of this scenario, the present study has described all aspects of the algorithm.

4 Proof of correctness

Proving the correctness of the proposed algorithm depends on satisfying safety and liveness properties. Therefore, the present work should separately prove that each of these properties is assured.

4.1 Safety

Safety is assured if no more than one node executes its CS at one time. 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-holder node in the current work's token-based algorithm. Of course, this node cannot remain the token-holding node because the proposed algorithm requires perpetual movement of the token. As only the token-holder node can enter its CS, it is sufficient to show that just one token-holder node exists at any given time. Only the token-holder node can send the token to just one other node, after which it becomes a non-token-holder node. The token is transferred to the receiver node within a limited time. Furthermore, a token cannot be produced and sent by any non-token-holder node.

Theorem 1 (Safety) *The proposed algorithm in Fig. 2 achieves safety.*

Proof The present study uses “reduction to the absurd” to prove safety assurance. Thus, it must be stated that the safety is not assured. As a result, two or more nodes can simultaneously execute their CSs. In the proposed algorithm, because only the token-holder node can enter its CS, the system must then be a multi-token one. Thus, these tokens existed in the system, to begin with, or some nodes produced tokens, or some non-token-holder nodes sent the token messages to other nodes, or the token-holder node could have sent the token to more than one node. Considering these explanations, the assumptions are impossible. Hence, a contradiction exists that proves the assumption that more than one node can simultaneously enter CSs is incorrect. As a result, safety is assured. □

4.2 Liveness

Liveness is assured if every CS request will eventually be responded to. Liveness includes freedom of deadlock and starvation.

Theorem 2 (Liveness) *The algorithm in Fig. 2 achieves liveness.*

Proof By means of a contradiction, the present study proves that liveness is assured. It is therefore assumed that the algorithm does not ensure liveness. This assumption can be the result of one of the following situations:

- None of the nodes are token-holder nodes, and *Token* cannot be forwarded to other nodes: This case is erroneous because it is mentioned, on the assumption, that P_k is the token-holder node at the beginning of the algorithm and *Token* is passed from one node to another.
- The token-holder node does not know whether other nodes have requests or not: This is incorrect because when a node attempts to invoke its CS, it inserts the *Request* in its *Waiting* queue and then sends the *Request* to the next node along the horizontal ring. This forwarding of the *Request* message is continued until it returns to the owner node. In the circular path of the *Request* message, each node also inserts the *Request* in its *Waiting*. Furthermore, *Token* circulates vertically and eventually meets up with one of these alerted nodes. Therefore, *Token* understands that there is at least one pending request in this row and so starts its horizontal movement to respond to any possible request. It is noteworthy that *Token* resumes its vertical circulation as it completes a horizontal rotation. As a result, there is no starvation of nodes in other rows because the new requests of a row do not permanently keep the *Token* in that row. Therefore, the first assumption is wrong.
- The token-holder node does not pass *Token* to other nodes and keeps it forever. This assumption is incorrect because if the token-holder node attempts to invoke its CS, it executes the CS in a finite time period. After releasing its CS and clearing its *Waiting*, the *Token* must be forwarded to another node. Therefore, in these two conditions, this node becomes a non-token holder node:
    1. If *Token* has not completed its horizontal circulation, it will be forwarded to the neighbor on the right of the token-holder node.
    2. Otherwise, *Token* will be delivered to the neighbor below to continue its vertical movement in a column.

This contradiction then indicates that the anti-liveness assumption cannot be corrected.

- Messages do not arrive at the destination node: This is incorrect. Based on the assumptions of the proposed algorithm, the network is error-free. Hence, this statement is also wrong.

In the end, liveness is assured. □

5 Comparison analysis

As deduced from the literature, the execution time of algorithm instructions is negligible in contrast to that of message passing. Therefore, similar to the other presented articles in the literature [28,38,40,42], the current study counts the number of messages exchanged in order to evaluate the performance of the algorithm. The performance of DME algorithms is considered under three conditions: light, medium, and heavy load situations [4,28,38,40]. In the literature, the overhead is often due to message passing. Therefore, the lower the number of message exchanges the lower the overhead of the algorithm. In addition, the scalability of DME algorithms depends on the message complexity of these algorithms. Therefore, the proposed algorithm (as shown in the following), due to reducing the message complexity, improves the scalability property and reduces the overhead.

5.1 Light demand situation

The present study assumed a status in which only process P_i (in row m and column n of the logical torus) attempts to invoke its CS. To do so, it inserts $Request_i$ in its *Waiting* and sends that message to its neighbor on the right. The neighbor also inserts the $Request_i$ in its *Waiting* and then forwards it to the node on the right. This action continues until $Request_i$ comes back to process P_i. This means that the message completely circulates a horizontal ring. Therefore, up to this stage of the algorithm, the number of messages exchanged is $\sqrt{N}$. In addition, the number of messages exchanged for *Token*, which are received by one of the alerted nodes in row m, is between 1 and $\sqrt{N}+1$. One message is required if *Token* is in row $m-1$; also, $\sqrt{N}+1$ is required when *Token* must circulate around a vertical ring and also change its column to arrive at a node in row m. After that, *Token* needs $\sqrt{N}$ relocations to respond to process P_i's request and other possible pending requests in this row. Consequently, as shown in Table 1, between $2\sqrt{N}+1$ and $3\sqrt{N}+1$, messages are exchanged under a light load condition. This is fewer than that of many other algorithms, such as [1,31–36,36–39].

5.2 Heavy demand situation

The current study assumes a status in which all nodes attempt to simultaneously invoke their CSs and in which each of them tries to enter the CS again immediately after releasing its CS. Therefore, each of these N nodes sends their requests to the next node on the right. Because each node places its *Request* message in its *Waiting*, a

Table 1 Evaluation of the algorithms

Algorithm	Broadcast-based/ Logical structure-based	Token-based/ Non-token-based	Average waiting time	Message complexity				
				Worst case	Average case	Best case	Heavy demand	Light demand
Le Lann [37]	Logical structure-based	Perpetual moving of the token	$O(N)$	N	$N/2$	1	1	$O(N)$
Suzuki–Kasami [34]	Broadcast-based	Token asking	$O(1)$	N	N	N	N	N
Raymond [39]	Logical structure-based	Token asking	$O(N)$	$2(N-1)$	$O(\log N)$	2	4	$O(\log N)$
Naimi–Trehel [36]	Logical structure-based	Token asking	$O(N)$	N	$O(\log N)$	2	–	$O(\log N)$
Lamport [1]	Broadcast-based	Non-token-based	$O(1)$	$3(N-1)$	$3(N-1)$	$3(N-1)$	$3(N-1)$	$3(N-1)$
Ricart–Agrawala [33]	Broadcast-based	Permission-based	$O(1)$	$2(N-1)$	$2(N-1)$	$2(N-1)$	$2(N-1)$	$2(N-1)$
Razzaque et al. [32]	Logical structure-based	Permission-based	$O(N)$	N	N	N	N	N
Maekawa [31]	Logical structure-based	Quorum-based	$O(1)$	$5(\sqrt{N}-1)$	–	$3(\sqrt{N}-1)$	$5(\sqrt{N}-1)$	$3(\sqrt{N}-1)$
Proposed algorithm	Logical structure-based	Perpetual moving of the token	$O(\sqrt{N})$	$3(\sqrt{N}+1)$	–	3	3	$2(\sqrt{N}+1)$ to $3(\sqrt{N}+1)$
Paydar et al. [22]	Logical structure-based	Hybrid	–	$4\sqrt{N}$	–	1	–	$4\sqrt{N}$
Taheri et al. [40]	Logical structure-based	Hybrid token-based	$O(\sqrt{N})$	$2\sqrt{N}$	–	$\sqrt{N}$	$\sqrt{N}$	$O(\sqrt{N})$

node finds its *Waiting* non-empty and, in the case of PCL_1, does not forward the other node's request to the next node. Therefore, there are N message exchanges for all nodes up to this step of the algorithm. It should be noted that, if some nodes asynchronously attempt to invoke their CSs, they just place that request in their *Waiting* and do not forward it to the next node according to PCL_2. The reason for this is that they find their *Waiting* non-empty. Thus, there are still at most N message exchanges in this case.

At this time, the number of messages exchanged by the *Token* movement must be counted. *Token* circulates a row to respond to all pending requests on that row, and there are $\sqrt{N}$ number of rows. Therefore, $\sqrt{N} \times \sqrt{N} = N$ messages to be exchanged for *Token* movements in rows. Also, $\sqrt{N} + 1$ messages are exchanged for the vertical circulation of *Token* in regard to column changing.

Totally, under heavy condition, there are $2N+\sqrt{N}+1$ message exchanges including *Token* and *Request* messages. Therefore, the average number of messages exchanged per CS invocation is, at the most, three:

$$\frac{2N + \sqrt{N} + 1}{N} \leq 3(\text{for } N > 2),$$

which is far less than that of many other algorithms (e.g., [1,31–34,36,39,40]), as shown in Table 1.

5.3 Medium demand situation

Here, a status is assumed in which $\sqrt{N}$ nodes attempt to simultaneously invoke their CSs. In this situation, two cases are considered:

- **Best case:** This case occurs whenever $\sqrt{N}$ requester nodes are located in the same row. Therefore, similar to the heavy demand situation, there are $\sqrt{N}$ message exchanges for all nodes in this row to request their CSs. On the other hand, at most, $\sqrt{N} + 1$ messages are required for *Token* to arrive at the alerted node in this row. Then, there are $\sqrt{N}$ message exchanges to move *Token* to this row. Totally, $3\sqrt{N} + 1$ messages are exchanged in this case; therefore, the average number of messages exchanged per CS invocation is approximately three.
- **Worst case:** This case presents a situation in which $\sqrt{N}$ requester nodes are located in different rows. This case is similar to the light demand situation in that, to request the CS, each requester node alerts the nodes in its rows; this requires $\sqrt{N} \times \sqrt{N} = N$ messages to be exchanged. Also, $\sqrt{N} + 1$ and $\sqrt{N} \times \sqrt{N} = N$ messages are exchanged for the vertical and horizontal circulation of *Token*, respectively. Therefore,

$$\frac{2N + \sqrt{N} + 1}{\sqrt{N}} \leq 3\sqrt{N} \text{ (for } N > 2)$$

Thus, the average number of messages exchanged per CS invocation (3 to $3\sqrt{N}$) is less than that of many other algorithms (e.g., [1,32–34,37]).

6 Conclusion and future work

This present paper presented a new distributed algorithm to solve the mutual exclusion problem in distributed environments. A logical structure was assumed in the form of a two-dimensional torus, in which requests are sent in a horizontal ring (rows of the torus), and a token rotates in a vertical ring (the columns of the torus). The current work proved that the algorithm correctly satisfies CS entry requests, and hence, safety and liveness properties are assured. By maintaining the simplicity of the implementation, the scalability property was increased and the average waiting time shortened in comparison with the token-ring algorithm. In addition, the overhead decreased due to the manipulation of high-volume data, in contrast to many other algorithms.

Generally, in light demand scenarios, the number of necessary message exchanges, between $2\sqrt{N}+1$ and $3\sqrt{N}+1$ per CS invocation, is more than that of heavy demand conditions (3 messages). In comparison with leading algorithms, the performance of the proposed algorithm is better in terms of fewer message exchanges, especially in the heavy load situations. Therefore, the authors recommend that the algorithm be implemented in distributed systems with high demand. For future work, focus will be placed on fault tolerance aspects, such as token loss, which is a significant issue for token-based DME solutions.

References

1. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
2. Saxena PC, Rai J (2003) A survey of permission-based distributed mutual exclusion algorithms. Comput Stand Interfaces 25(2):159–181
3. Thapliyal H, Arabnia HR, Srinivas MB (2009) Efficient reversible logic design of BCD subtractors. In: Gavrilova ML, Kenneth Tan CJ (eds) Transactions on computational science III. Springer, Berlin, pp 99–121 **(LNCS 5300)**
4. Lejeune J, Arantes L, Sopena J, Sens P (2015) A fair starvation-free prioritized mutual exclusion algorithm for distributed systems. J Parallel Distrib Comput 83:13–29
5. Arabnia HR, Oliver MA (1986) Fast operations on raster images with SIMD machine architectures. Comput Graph Forum 5(3):179–188
6. Thapliyal H, Jayashree HV, Nagamani AN, Arabnia HR (2013) Progress in reversible processor design: a novel methodology for reversible carry look-ahead adder. In: Gavrilova ML, Kenneth Tan CJ (eds) Transactions in Computational Science. Springer, pp 73–97 **(LNCS 7420)**
7. Thapliyal H, Arabnia HR, Bajpai R, Sharma KK (2007) Combined integer and variable precision (CIVP) floating point multiplication architecture for FPGAs. In: Proceedings of 2007 International Conference on Parallel and Distributed Processing Techniques and Applications; PDPTA'07. pp 449–450
8. Thapliyal H, Arabnia HR (2006) reversible programmable logic array (RPLA) using Fredkin and Feynman gates for industrial electronics and applications. In: Proceedings of 2006 International Conference on Computer Design and Conference on Computing in Nanotechnology (CDES'06). pp 70–74
9. Thapliyal H, Srinivas MB, Arabnia HR (2005) Reversible logic synthesis of half, full and parallel subtractors. In: Proceedings of 2005 International Conference on Embedded Systems and Applications, ESA'05. pp 165–172
10. Qureshi MB, Alrashed S, Min-Allah N et al (2015) Maintaining the feasibility of hard real-time systems with a reduced number of priority levels. Int J Appl Math Comput Sci 25(4):709–722. doi:10.1515/amcs-2015-0051

11. Alrashed S, Alhiyafi J, Shafi A, Min-Allah N (2016) An efficient schedulability condition for non-preemptive real-time systems at common scheduling points. J Supercomput 72:4651–4661. doi:10.1007/s11227-016-1751-6
12. Hura GS (1989) The role of atomic actions in a distributed system. Microelectron Reliab 29(2):185–193
13. Leiserson CE (2016) A simple deterministic algorithm for guaranteeing the forward progress of transactions. Inf Syst 57:69–74
14. Tamhane SA, Kumar M (2012) A token based distributed algorithm for supporting mutual exclusion in opportunistic networks. Pervasive Mob Comput 8:795–809
15. Jiang J-R (2011) Nondominated local coteries for resource allocation in grids and clouds. Inf Process Lett 111:379–384
16. Bertier M, Obrovac M, Tedeschi C (2013) Adaptive atomic capture of multiple molecules. J Parallel Distrib Comput 73:1251–1266
17. Aravind AA (2013) Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms. J Parallel Distrib Comput 73:1029–1038
18. Wu W, Zhang J, Luo A, Cao J (2015) Distributed Mutual Exclusion Algorithms for Intersection Traffic Control. IEEE Trans Parallel Distrib Syst 26:65–74
19. Ni W, Wu W, Li K (2016) A message efficient intersection control algorithm for intelligent transportation in smart cities. Futur Gener Comput Syst. doi:10.1016/j.future.2016.10.033
20. Ciuffoletti A (2010) The wandering token: congestion avoidance of a shared resource. Future Gener Comput Syst 26(3):473–478
21. Tanenbaum AS, Van Steen M (2007) Distributed systems-principles and paradigms. Pearson Prentice Hall, Upper Saddle River
22. Paydar S, Naghibzadeh M, Yavari A (2006) A hybrid distributed mutual exclusion algorithm. In: International Conference on Emerging Technologies 2006, ICET'06. IEEE, pp 263–270
23. Dijkstra EW (1965) Solution of a problem in concurrent programming control. Commun ACM 8:569
24. Li Z, Yan M, Zhou M (2010) Synthesis of structurally simple supervisors enforcing generalized mutual exclusion constraints in petri nets. IEEE Trans Syst Man Cybern C (Appl Rev) 40:330–340
25. Hesselink WH (2016) Correctness and concurrent complexity of the Black-White Bakery Algorithm. Formal Asp Comput 28(2):325–341
26. Bienkowski M, Klonowski M, Korzeniowski M, Kowalski DR (2016) Randomized mutual exclusion on a multiple access channel. Distrib Comput 29(5):341–359
27. Taubenfeld G (2014) Tight space bounds for ℓ -exclusion. Distrib Comput 27(3):165–179
28. Park S-H, Lee S-H (2014) Quorum-based mutual exclusion in asynchronous distributed systems with unreliable failure detectors. J Supercomput 67(2):469–484
29. Aravind AA (2010) Highly-fair bakery algorithm using symmetric tokens. Inf Process Lett 110:1055–1060
30. Kakugawa H (2015) Self-stabilizing distributed algorithm for local mutual inclusion. Inf Process Lett 115:562–569
31. Maekawa M (1985) An algorithm for mutual exclusion in decentralized systems. ACM Trans Comput Syst 3(2):145–159
32. Razzaque MA, Hong CS (2008) Multi-token distributed mutual exclusion algorithm. In: 22nd International Conference on Advanced Information Networking and Applications, 2008, AINA. IEEE, pp 963–970
33. Ricart G, Agrawala AK (1981) An optimal algorithm for mutual exclusion in computer networks. Commun ACM 24(1):9–17
34. Suzuki I, Kasami T (1985) A distributed mutual exclusion algorithm. ACM Trans Comput Syst 3(4):344–349
35. Raynal M (1991) A simple taxonomy for distributed mutual exclusion algorithms. ACM SIGOPS Oper Syst Rev 25(2):47–50
36. Naimi M, Trehel M, Arnold A (1996) A log (N) distributed mutual exclusion algorithm based on path reversal. J Parallel Distrib Comput 34(1):1–13
37. Le Lann G (1977) Distributed systems-towards a formal approach. In: IFIP Congress. Toronto, pp 155–160
38. Neamatollahi P, Taheri H, Naghibzadeh M (2012) Info-based approach in distributed mutual exclusion algorithms. J Parallel Distrib Comput 72(5):650–665
39. Raymond K (1989) A tree-based algorithm for distributed mutual exclusion. ACM Trans Comput Syst 7(1):61–77

40. Taheri H, Neamatollahi P, Naghibzadeh M (2011) A hybrid token-based distributed mutual exclusion algorithm using wraparound two-dimensional array logical topology. Inf Process Lett 111:841–847
41. Agrawal D, El Abbadi A (1991) An efficient and fault-tolerant solution for distributed mutual exclusion. ACM Trans Comput Syst 9(1):1–20
42. Kakugawa H (2015) Mutual inclusion in asynchronous message-passing distributed systems. J Parallel Distrib Comput 77:95–104