International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-8 Issue-4, April 2019

An Improved Token-based Distributed Mutual Exclusion Algorithm

Prashant Kumar, Naveen Kumar Gupta, Rama Shankar Yadav, Rajendra Kumar Nagaria

***Abstract:** In a distributed system, access to shared resources in a mutual exclusive manner has always been a challenging problem. Mutual exclusion is required to share the resources among multiple processes so that only one process can access that shared resource at a time. Thus, no race condition occurs. To resolve this problem, we present an improved token-based distributed mutual exclusion (ITDME) algorithm which provides access of shared resources in mutual exclusive manner with reducing time complexity. The algorithm works on torus topology which is a logical structure on top of the physical structure. The proposed algorithm outperforms existing distributed token-based algorithm in terms of average running time.*

***Index Terms**: Distributed System, Mutual Exclusion, Torus Topology.*

I. INTRODUCTION

A distributed system is a collection of autonomous computers connected through a communication network that appears to its user as a single coherent system[1]–[2]. There is no common memory, they communicate with each other through message passing. In systems with multiple concurrent processes, it is economical to share resources among concurrently executing process. In such an environment, where several processes try to access shared resources simultaneously, consistency of resources is critical. For example, a file must not be updated simultaneously by several processes. In the absence of consistency, a race condition could occur in the system which leads to erroneous results. Hence, every process should access resources exclusively. The Exclusiveness of access is known as mutual exclusion, and the part of a program which needs exclusive access to the shared resource is called critical section (CS). Over the past several years, many MUTEX algorithms have been proposed [3]-[9]. The origin of the MUTEX problem first arose in a single processor system to provide exclusive access of the shared resource to the process [10]. This problem also occurred in a distributed environment where several processes from different systems try to access shared resources. Several algorithms have been proposed to solve the MUTEX problem in distributed systems. These algorithms are broadly divided into two categories [11]-[12]: permission-based algorithms [8], [13]-[17] and token-based algorithms [5], [9], [18]-[21]. In Permission-based algorithms, any node that wants to enter in its CS requests to all other nodes present in the system. On receiving the request message, nodes reply to the requestor node. After getting a response from all nodes, based on the response it decides whether it could execute its CS or not. The primary issue with these algorithms is high communication overhead. In token-based MUTEX algorithms, a token is shared among all nodes of the distributed network. The token moves from one node to another. Whenever a node wants to execute its CS, it waits until it receives the token. Upon receiving a token, it executes its CS and then sends the token to some other node. Hence, only one node can execute its CS at a time. Any MUTEX algorithm should ensure the following properties:

1. Safety property: At any instant of time, only one process can execute its CS.
2. Liveness property: Two or more processes should not wait for any message indefinitely. A node should get a chance to execute its CS in a definite time.
3. Fairness: Each process should get a chance to execute its CS.

The rest of the paper is organized as follows. Section 2 contains a literature survey which acknowledges previous work in the proposed field. Section 3 describes the proposed work with required algorithms. In section 4, we have discussed the simulation and results. Finally, section 5 concludes the paper.

II. RELATED WORK

In the past three decades, several distributed algorithms were developed. Some are permission-based [5], [8], [13]and some are token based [5], [9], [21]-[22]. In permission-based algorithms, the node enters into CS aftertaking permission from othersites present in the system. These algorithms are broadly divided into two categories, based on the timestamp and voting scheme. On the other hand, in token-based algorithms, a privileged message called token is passed from one node to another. Any node can enter in CS if it holds the token.

Manuscript published on 30 April 2019.
* Correspondence Author (s)
Prashant Kumar, Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology Allahabad, Prayagraj, India.
Naveen Kumar Gupta, Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology Allahabad, Prayagraj, India.
Rama Shankar Yadav, Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology Allahabad, Prayagraj, India.
Rajendra Kumar Nagaria, Department of Electronics and Communication Engineering, Motilal Nehru National Institute of Technology Allahabad, Prayagraj, India.

These algorithms use sequence number for handling outdated requests which are already fulfilled. These algorithms differ from each other in the way they find the token in the system. Token-based algorithms have fewer messages per CS execution because of the existence of a token in the system. Suzuki-Kasami[21] proposed a token-based mutual exclusion algorithm in which requesting node sends the request message to all other nodes. After that, it waits for token to execute CS. Each node stores the latest sequence number of the request generated by each node. In the token message, a queue is maintained, which stores the pending request and an array of the sequence number of last executed requests of each node. After executing CS, the node sends the token to the first node in the queue. It requires at most N messages for completing one CS request. The drawback of the algorithm is that the sequence number is unbounded. In [9], the author used a heuristic approach which helps to reduce the number of messages required per CS invocation. It keeps the state information of each node for changing its request set. A node could be in one of the following states: Requesting, Not Requesting, Executing and Holding. The algorithm uses sequence number for determining old and new request. The fairness of the algorithm depends upon the arbitration rule, which is used by a node to send the token to the next node after executing its own CS. The space requirement of the algorithm is quite large because node and token, both store the state for each node. Chang et al.[23] proposed another MUTEX algorithm in which the request set of the node is changed dynamically. The token contains a request queue. Each node maintains a request set and an array of the sequence number. After executing CS, the node adds all pending requests present at that node in token queue. Simulation result shows that 60% of the number_of_nodes messages are required to execute CS under light load. However, during high load algorithm performed similar to Suzuki Kasami's algorithm, i.e., the number of messages is equal to the number of nodes. Yan et al. [24] algorithm uses dynamic state information. Each node dynamically updates the latest location of the token. Any node which wants to execute CS, sends the request to nodes according to latest known location. As information present locally at each node changes, the path of the request also changes accordingly. Total message complexity of the algorithm isO(N)where N is the number of nodes. Saxena et al. [25] presented an approach in which the following states of the node are possible: Requesting, Not Requesting, Executing and Holding. The token fulfils all those requests which fall on the route to the destination node. However, the algorithm does not follow FCFS order, but in case of heavy load, response time and message complexity are both reduced considerably. The algorithm also uses a special kind of message, token location propagator (TLPS). This message is sent to other nodes when node releases the token. Nowadays, joining and releasing of the nodes is very frequent in a distributed system. In such dynamic scenarios, maintaining the topology is difficult. Hence, applications of this algorithm are very limited. Neamatollahi et al. [5] proposed a new token-based algorithm which uses torus topology as a logical structure. Nodes are divided into an equal number of rows and columns where CS request moves in the horizontal ring while token moves in the vertical ring. Whenever any node wants to execute its CS, it sends its CSRequest to its right neighbour. This way all nodes in a row know about pending requests. When the token reaches to any of these nodes, during vertical movement, the node sends the token in the horizontal direction to complete pending CS request in that row. When the token reaches the starting node, it again resumes its vertical movement. The drawback of this algorithm is that when the requestis made by a left neighbour of the token holding node, it still reaches to that node by completing its horizontal movement. Because of this, the response time of these requests increases.

III. PROPOSED WORK

We have proposed an improved version of Neamatollahi et al.'s algorithm, named as Improved Token based Distributed Mutual Exclusion (ITDME). In [5], when the token started its horizontal movement, it moves from right to left to cover each node, whether the node requested for CS or not. Suppose node k requested for CS, which is just next to the token holder node. So, to complete node k's request, token travel all the nodes from the right direction. Due to which response was increased.

To solve this issue, we have proposed a modification in the algorithm. When a node receives token, it sends the token to the nearest CS requesting node, instead of sending token blindly to right direction. Besides this, we also send a request message from both directions, i.e., left and right, so that the time required to reach the request message among nodes present in the horizontal ring gets reduced.

A. Model and Assumptions

The present algorithm is implemented on a distributed system having N nodes numbered 0, 1, ..., N-1 without having any shared memory. These nodes communicate with each other through message passing. We make the following assumption about the network:

1. The communication network is error free.
2. Without loss of generality, we assume that there is only one process at each node. Hence, process and node can be used interchangeably.
3. The propagation delay is unpredictable, but finite. It means that every node will get the message eventually.
4. The message could be received in a different order than in which it is sent.
5. A unique identification number is given to each process between 0 to N-1.
6. We assumed that $N = k^2$, where k is an integer and N, is the number of nodes in the system.

In broadcast algorithms, CS request is broadcasted to all other nodes in the system. Thus, it does not require any logical configuration. However, in the logical structure-based algorithm, the nodes are arranged in some logical form like a ring or tree. We have used two-dimensional torus topology as a logical structure in which every node is part of two rings, horizontal and vertical ring.

It is supposed that critical resource can be accessed by the process only when the process is executing its CS and CS execution time is finite. The process cannot request for another CS before its previous request gets executed.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

B. Data Structures and Messages

Data structure and message used in the proposed algorithm are as follows:

1. **The token** is a control message, it has the following fields:
   a. Row: It is used to count the number of rows token passed. When token completed one vertical movement, then token sends to next node *S*, present in the right direction and value of row set to be 0. So that *S* could start a vertical movement of the token in its column.
   b. nextExecProcess: It is used to save the node id which is going to execute its CS.
   c. Direction: It is used to save the direction movement of the token. It could be left, right and down.
   d. CSSeq: It is an array of size N, where N is the number of nodes present in the system. It stores the sequence number of each CS request that got executed for each node. Hence, CSSeq[i] is a number of CS request of node *i* executed till now.
   e. CSExec: It is a set which has at most$\sqrt{N}$ elements. In any particular row when any node executes its CS, it adds its node Id in this set. It is used to handle the starvation condition. At any particular node, when it releases the token, it checks the size of CSExec. If its size is equal to a number of nodes present in the row, then token will be sent in down direction whether any pending request is present or not.
2. **The request** is a control message, which contains the following fields:
   a. Node: Identification number of the node which requested for CS.
   b. SeqNo: Sequence number of current CS request for a node whose identification number is saved in node field.
   c. Distance: It stores the distance of given node from the current node.
3. Request Message is sent by a process which wants to execute its CS. It consists of the identification number of the node, the sequence number for this request and direction in which request will be sent.
4. Each node has the following fields:
   a. SeqNo: It is used to track the current CS request. Whenever a node wants to execute CS, it increments SeqNo by 1 and then sends its request to its neighbour in the horizontal ring.
   b. CSPermission: It determines whether the process can enter in its CS or not.
   c. ReqArr: It is an array of Request record. Its size is 3. It stores the nearest left request at 0^{th} position, its request at 1^{st} position, and its nearest right request at 2^{nd} position.

C. Proposed Approach

All nodes are connected to its neighbour according to torus logical structure. As long as nodes are idle, the token will move in its vertical ring starting from node *j*. After completing one cycle, i.e., token again reached to node *j*, token move to the right column and traversed all nodes of the column. After covering all columns of the network, the token will return to node *j*. If any node *k* turns from idle to waiting after requesting for its CS, it sends it to request with the sequence number to both of the neighbours in the horizontal ring, i.e., left and right node. After receiving a request from another node, if it did not request for CS, then it will forward the request to its neighbour according to the direction of the message. It will compute the distance of node *k* from itself, and according to computed distance and after checking the value of ReqArr at the 0^{th} and 2^{nd} position, it will save the request or discard it. This way nearest node on the horizontal ring in both direction aware about CS request of node *k*. Eventually during its column movement token will reach to any one of these nodes, and then the token will be sent to node *k*. It could be from the left side or right side.

D. Description of Proposed Approach

Let's suppose, node 2 is a token holding process. The token is moving in the downward direction in column 2. Let's assume node15 is the only requestingnode which wants to execute its CS. The location of these nodes is shown in Fig. 1, node 15 sends its CS request in both directions, i.e., right and left.

When node 16 and node 19 received request message of node 15, then they compute the distance of node 15 and itself from the right and left direction. Since node 15 is the nearest node from node 16 from the left direction, hence it saves the request message at the 0^{th} position of its local ReqArr. Similarly, for node 19, node 15 is nearest from the right direction. Thus, it saves the request message at 2^{nd} position of its local ReqArr. If node 16 and node 19 do not want to execute its CS then they will forward therequest to their neighbour according to the direction of the message.

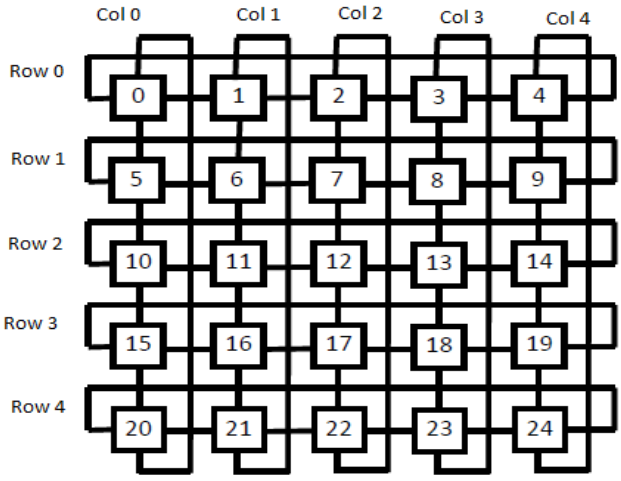

Fig. 1. Torus Topology

When node 17 receives the token, it saves the node 15's CS request. Hence token will be sent to node 15 from the left direction after changing NextExecProcess and direction field of a token. Suppose meanwhile node 16 also requested for CS, when token reached to node 16. Hence it will execute its CS and then send the token to node 15 after changing its SeqNo in CSSeq field in the token, without deciding the direction of the token.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

In this way during horizontal movement, when nextExecProcess and direction is fixed in the token, still, any node which comes in the path can also execute the CS. It will help to reduce the response time of some CS Requests.

E. Algorithm Details

The presented algorithm can be divided into three parts: In the first part, the node requests for the critical section. In the second part, the node receives a message from another process and execute its CS. In the third part, the node will release the CS. Initially, each node executes algorithm 1 for initialization.

Algorithm 1 Initialization of token and local fields

```
1: token.row =0, token.CSExec ={}, token.CSSeq[N] ={0},
   direction = down, nextExecProcess = 0
2: Request = {node, seqno, distance}
3: for all Processes: CSPermission = false, seqNo = 0,
   RequestReqArr[3]
```

Algorithm 2 Requesting the CS

```
1: function REQUESTCS()
2:    seqNo_i++;
3:    ReqArr[1] = {i,seqNo_i, 0}
4:    send message {node :i,seqNo : seqNo, direction : left}
      to left neighbor
5:    send message {node :i,seqNo : seqNo, direction :
   right}  to right neighbor;
6:    WAIT(CSPermissioni = TRUE)
7: end function
```

Requesting the CS

Suppose node *p* want to execute the CS then it will follow the procedure as shown in algorithm 2. According to algorithm 2, node *p* will increase its sequence number and create a request message with its identification number and sequence number. After that, it will insert own request in its ReqArr before sending this request message to its left and right neighbour in the horizontal ring. Then, node *p* waits until it receives the token. Whenever node *p*receives the token, it sets CSPermission to TRUE and executes its CS. After that, it includes its identification number in CSExec set of token and update its sequence number in CSSeq array of the token.

Algorithm 3 Check for any pending request is present on node or not

```
1: function isANYPENDINGREQUEST()
2:    nextRequest = -1
3:    if token.CSExec.size =√N then
4:       nextRequest = -1
5:    else if ReqArr[0] != NULL &&ReqArr[2] != NULL
   then
6:       if ReqArr[0].node ∉token.CSExec&&
         ReqArr[2].node∉token.CSExec then
7:          if ReqArr[0].distance >ReqArr[2].distance then
8:             nextRequest = 2
9:          else
10:            nextRequest = 0
11:         end if
12:      else if ReqArr[0].node ∉token.CSExec then
13:         nextRequest = 0
14:      else
15:         nextRequest = 2
16:      end if
17:   else if ReqArr[0] != NULL &&ReqArr[0].node ∉
      token.CSExec then
18:      nextRequest = 0
19:   else if ReqArr[2] != NULL &&ReqArr[2].node ∉
      token.CSExec then
20:      nextRequest = 2
21:   end if
22:   return nextRequest;
23: end function
```

Algorithm 5 Handle incoming request message

```
1: functionhandleREQUESTMESSAGE(msg)
2:    canForwardRequest = false
3:    if i != msg.node then
4:       if ReqArr[1] == null &&ReqArr[0]!=
         {msg.node,msg.seqNo}&&ReqArr[2] !=
         {msg.node,msg.seqNo} then
5:          canForwardRequest = true
6:       end if
7:       leftDistance = 0
8:       rightDistance = 0
9:       if i<msg.node then
10:         rightDistance = msg.node - i - 1
11:         leftDistance =√ N - rightDistance - 2
12:      else
13:         leftDistance = i - msg.node - 1
14:         rightDistance =√N - leftDistance - 2
15:      end if
16:      if leftDistance>rightDistance then
17:         if ReqArr[2] == NULL ||ReqArr[2].node >=
            msg.node then
18:            ReqArr[2] ={message:node,message:seqNo,
            rightDistance}
19:         end if
20:      else
21:         if ReqArr[0] == NULL ||ReqArr[0].node <=
            msg.node then
22:            ReqArr[0] ={message:node,message:seqNo,
            leftDistance}
23:         end if
24:      end if
25:      if canForwardRequest = TRUE then
26:         Send msg to msg.direction neighbor
27:      end if
28:   end if
29: end function
```

Receiving a Message

When node p receives a message from node k, it could be one of these two messages:

1. **Request Message:** Whenever node p receives a request message from node q. It extracts Nodes' identification number, sequence number and direction from the message. If ReqArr did not contain this request message and node p did not request for CS, then request message will be forwarded to next neighbour according to the direction of the request message. Node p computes the distance between itself and node q from both directions, i.e. left and right. If node q is the nearest from the left side, then it checks 0^{th} position of ReqArr. If it is empty ornode qis the nearest compared to the previous entry, then it will update the 0^{th} position entry in ReqArr.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

Similarly, if node q is nearest from the right side, then it checks 2nd position of ReqArr. If it is empty or node q is nearest compared to the previous entry, then it will update 2nd position entry in ReqArr.

2. This way at least two nodes from the left and right direction knows about the pending request of node q for CS execution.
3. **Token Message:** When node p receives a token message, it may come from three directions left, right and above. If the token is received from the neighbour above of node p, it enters in a new row. Hence row field of token gets incremented by one.

Algorithm 6 Handle incoming token messages

```
1: function handleTOKENMESSAGE(token)
2:    if token.direction = down then
3:       token.row = token.row+1
4:    end if
5:    updateReqSet();
6:    if token.CSSeq[i] <seqNo_i then
7:       CSPermissioni = TRUE
8:    else
9:       releaseCS()
10:   end if
11: end function
```

After that, by comparing the CSSeq field of token node p updates its ReqArr.It is required because it may be possible that node p is not aware of that node q already execute the CS for the current sequence number. Hence this way, if any outdated request is present in ReqArr, then it will be removed. If node p also requested for CS, then it will execute its CS otherwise if nextExecProcess field value of the token is not the identification number of node p, then it sends the token to its neighbour according to direction field of token. However, if the nextExecProcess value is the identification number of node p, then it will check 0th and 2nd position entry in ReqArr and find the nearest node which is not part of the CSExec field of a token. If it finds out such pending request, then set NextExecProcess and direction field accordingly and sends the token to left or right neighbour accordingly.

In case, if there is no pending request present, if the token completes its vertical movement for current column, then token will be sent to the next right neighbour after setting row as *0*, direction as right, clearing the CSExec field and set NextExecProcess to right neighbour. Otherwise token is sent to downneighbour after setting NextExecProcess and clearing CSExec field.

Algorithm 7 Update ReqArr of node by using token.CSSeqarray

```
1: function updateREQSET()
2:    if ReqArr[0] != NULL &&ReqArr[0].seqNo<=
      token.CSSeq[ReqArr[0].node] then
3:       ReqArr[0] = NULL
4:    end if
5:    if ReqArr[2] != NULL &&ReqArr[2].seqNo<=
      token.CSSeq[ReqArr[2].node] then
6:       ReqArr[2] = NULL
7:    end if
8: end function
```

Releasing the CS

When node *p* finishes its CS execution, it sets *CSPermission* to False, add its sequence number to *CSSeq* field of token, add its identification number in *CSExec* field of token. After that, if the *nextExecProcess* field value of the token is not the identification number of node *p* then it sends the token to its neighbour according to direction field of token else it checks for any pending request. If it found any request, then send token to that node else sends token to right or down neighbour after checking row field of token.

Algorithm 8 Releasing the Token

```
1: function RELEASETOKEN()
2:    if token.nextExecProcess = i then
3:       request = isAnyPendingRequest()
4:       if request = -1 then
5:          CLEAR token.CSExec
6:          if token.row =√N then
7:             token.row = 0;
8:             token.nextExecProcess = right neighbor
9:             send token to right neighbor
10:         else
11:            token.nextExecProcess = down neighbor
12:            send token to down neighbor
13:         end if
14:      else
15:         token.direction = (request = 0 ? left : right)
16:         token.nextExecProcess = ReqArr[request].node
17:         send token to token.direction neighbor
18:      end if
19:   else
20:      send token to token.direction neighbor
21:   end if
22: end function
```

Algorithm 3 Releasing the CS

```
1: function RELEASECS()
2:    CSPermissioni = false;
3:    token.CSSeq[i] = seqNo_i;
4:    token.CSExec = token.CSExecUi;
5:    ReqArr[1] = NULL
6:    releaseToken()
7: end function
```

A. A Scenario

In this section we used a scenario to explain the algorithm in detail. Node *0* is the current token holding node and send the token in a downward direction while node *19* attempt to invoke its CS.

As shown in figure 2.a, the token is present at node *0*. Since node *0* does not want to execute its CS and there is no pending request present in its ReqArr, hence it will send token to node *5*. Node *19* attempted to invoke its CS. So, it sends a request message to node *15* and node *18*. Node *18* saves request message at 2nd position in its ReqArr while node *15* save it on 0th position. Since both the node does not want to execute CS therefore, forward this request message to their neighbour. When node *15* receives the token, it checks for any pending CS request for itself. If it does not want to execute CS then it checks its ReqArr for any pending request. Its ReqArr contains node *19's* request message. Since node *19* is the nearest to node *15* from the left side so node *15* send token to node *19* through left direction (see Figure 2.b).

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

As depicted in Figure 2.c, during execution of CS at node *19* and node *21* also requested to execute the CS.
Thus, it increments its sequence number and sends the request message to its left and right neighbours. Node *20* and node *22* save the request message at $2^{nd}$and 0^{th} position respectively. Similarly, node *24* and node *23* save the request message at 2^{nd} and 0^{th} position respectively. After executing CS, node *19* does not have any pending request so it will send token to node *24*.

Since, node *24* does not want to execute its CS and its ReqArr contains node *21's* request. So, it will set the value of *token.nextExecProcess* to node *21*, direction to right and send it to node *20* because distance of node *21* from node *24* is less from right direction compared to left direction. When node *20* receives token, it forwards it to node *21* according to *token.direction* field (see in Figure 2.d). After executing CS, node *21* checks its *ReqArr*. There is no pending request present, so it will send the token in a downward direction as shown in Figure 2.e. Finally, the token will reach to node *1*.

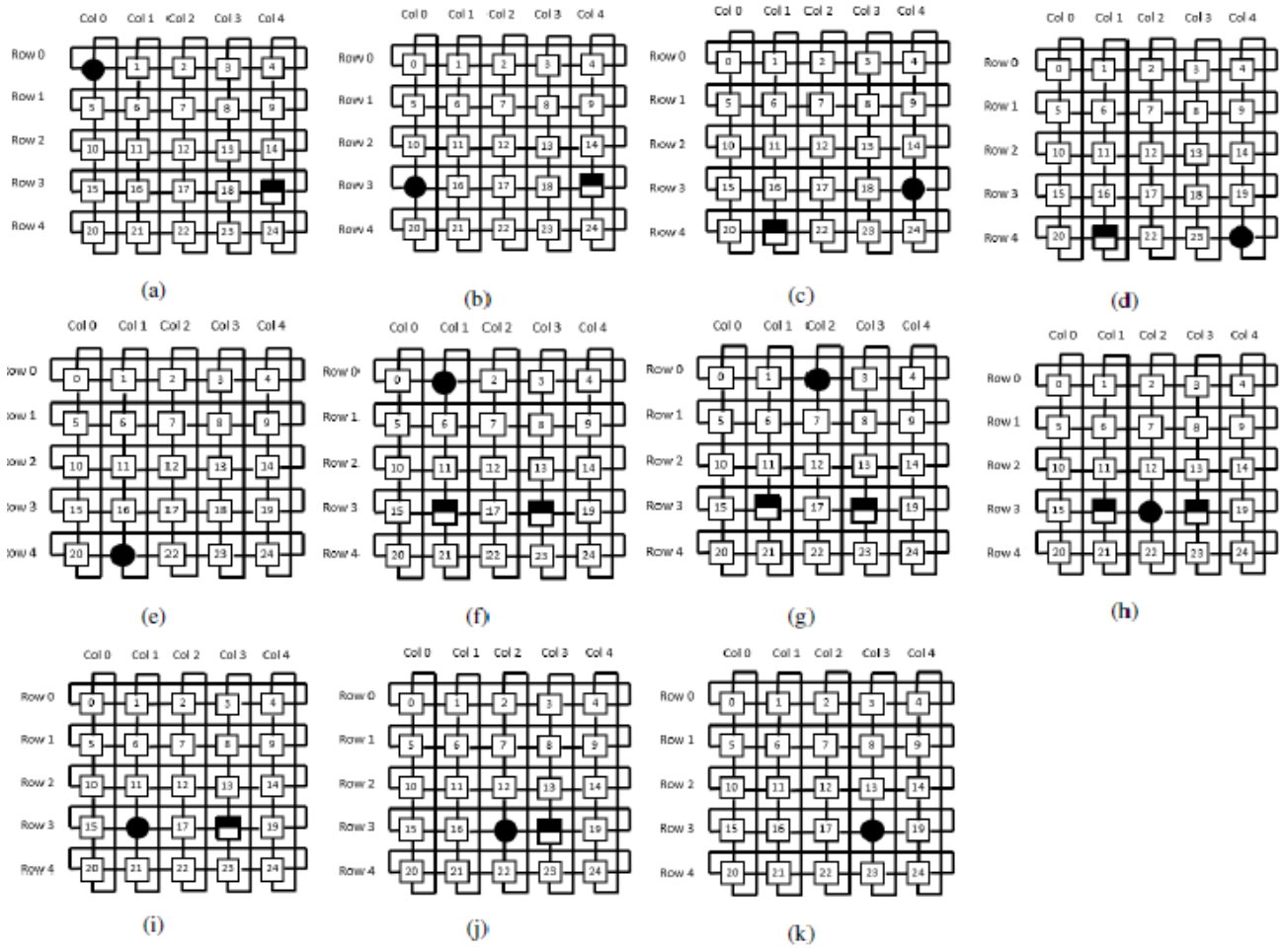

Fig. 2. Scenario

As shown in Figure 2.f, when token reached to node *1*, it completed its vertical cycle. Now, node *1* will send the token to node *2* which is the right Neighbour of node *1*. Meanwhile node *16* and node *18* also requested for CS. Node *17* saves node *16's* request at the 0^{th} position in ReqArr while node *18's* request at 2^{nd} position. Similarly, node *18* saves node *16'*srequest at the 0^{th} position in its *ReqArr*. Since node *18* also requested for CS, so it will not forward the node *16's* request to node *19* because node *18* is near to node *19* compared to node *16*. Similarly, node *16* will not forward node *18's* request to node *15* also.

In Figure 2.g, node *2* received token from node *1*. Neither node *2* wants to execute the CS nor it has any pending request, then it will send the token to downward node i.e. node *7*. Same condition will occur with node *7* and node *12* also. In this waytoken will reach at node *17*.Now, Node *17* receives token from node *12* (Figure 2.h). The node *17* has pending requests at both 0^{th} and 2^{nd} positions, therefore it check the distance from both nodes from itself. Here, distance is also same, so node *17* have to send the token to node *16*.

In Figure 2.i, node *16* receives token from node *17* and execute its CS. Since node *16* have pending requests at the 2^{nd} position, hence it sets *token.nextExecProcess* to node *18* and *token.direction*to right and send it to node *17* again. Node *17* receives token from node *16* as depicted in Figure 2.j, and after checking its ReqArr send token to node *18*. Node *18*receives token from node *17* and execute its CS. There is no pending request in its *ReqArr*, so it will send the token in a downward direction as shown in Figure 2.k.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

B. Proof of Correctness

Any distributed mutual exclusion algorithm should satisfy the safety and liveness properties. Hence, in this section, we proved that these properties are valid.

Safety

We use "reduction to the absurd" for proving that safety is assured in ITDME. Hence, we stated that safety is not guaranteed. Due to this statement, two or more than two nodes can execute their CS simultaneously. In ITDME, a node which has token can only execute its CS, so there should be more than one token present in the system. It means that either some node generated the token or node which do not have token, send the token to another or node which have a token, sent the token to more than one node. Based on this explanation, all assumptions are impossible and hence contradiction exist, which prove that at any instant of time two or more than two nodes can execute their CS. Thus, safety is guaranteed.

Liveness

To prove the liveness property, we use the contradiction. Therefore, we assume that ITDME does not guarantee liveness. Hence the following situation could be generated because of this assumption.

- The token does not exist in the system and cannot be sent to other nodes. This case is wrong because at the beginning of the algorithm, token is present at any one of the nodes in the system and the token will be passed from that node to another node.
- The token-holder node does not have any information that any pending request present in its row or not. This is incorrect because when any node wants to execute its CS, it saves the request at 1st position in ReqArr. After that it forward the request message to its left and right neighbour. Next node will save this request to either on 0^{th} or 2nd position of ReqArr. If theyalready requested for CS, then they do not forward the request message because for other nodes these nodes are the nearest node compared to the current requesting node. This way in horizontal ring, each node has information about the nearest left and right requesting node. During its vertical movement when token reached to any of these nodes, token start its horizontal movement. Whenever any node executes its CS, it inserts its identification number into the CSExec field of token. This way during horizontal movement, we prevent the starvation condition. If at any node, pending request is from one of the nodes whose identification number is present in CSExec field, then instead of sending token to that node, current node sends the token in a downward direction. Hence our assumption is wrong.
- The token-holder node does not send the token to another node and keep it for an indefinite time. This assumption in wrong because when any node wants to execute its CS, after getting the token, it executed the CS in finite time. After executing the CS, the token must be forwarded to another node. Here two conditions could occur.
  - If there is any pending request present at the current node, then token will be sent to that requesting node in horizontal direction.
  - Otherwise, token will be sent to downward neighbour and it will resume its vertical movement.

This contradiction shows that anti-liveness assumption cannot be valid. Hence liveness is guaranteed.

IV. SIMULATION RESULTS

In this section, we have described the tools, simulation environment and parameters used in our research work for analysis of ITDME algorithm with respect to Neamatollahi et al.'s algorithm. We have analyzed following types of simulation effects and shown the performance of protocols using a bar graph.

1. **Number of Processes:** We vary the number of processes present in the system to check the scalability of the algorithms. As N increases, and assuming the mean arrival time is not changed, there are more requests for the CS. This will increase the waiting time [11].
2. **Arrival Rate:** Arrival rate is the time between generating two requests by a process. This parameter is exponentially distributed with λ as the mean. As processes begin requesting more furiously there will be more requests, the probability of concurrent request is higher and hence there is a reduction in the number of messages per CS access[26].

A. Performance Metrics

In this paper, following performance metrics are used for the analysis of ITDME and Neamatollahi et al.'s algorithm.

Message Traffic: It is the average number of messages exchanged among the nodes per CS execution [26].

Time Delay: It is the average time delay in granting the CS, which is the period of time between the instant a node involves mutual exclusion and the instant when a node enters the CS [26].

B. Simulation Setup and Parameters

Java is used as a programming language to implement ITDME and Neamatollahi et al.'s algorithm. The machine used for simulation is Intel i7 with 4GB of RAM. We created a log server, which is used for collecting statistics. Each node creates logs for all the incoming and outgoing messages. With the help of these logs, we compute the total number of messages used to execute CS. We collected the statistics for 100 CS requests per node. For each simulation run, statistics collected for initial 5% request were discarded to eliminate the effect of start-up. Following simulation parameters are used in this simulation.

Table 1. Simulation Parameters

Parameter Type	Value
Number of Processes	25,36,49,64,81,100
Message Propagation Delay	1 ms
Arrival Rate	0.1, 0.2, 0.3, ..., 1.0

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)

C. Results

Effect of Arrival Rate on a 25-Node System:

As shown in table 2, Neamatollahi's Algorithm takes approx. 3 messages under heavy load condition, which verify his analytical result for a heavy load condition.

ITDME uses more messages compared to Neamatollahi's Algorithm because every node sends a request message to all other nodes present in a horizontal ring under light load condition. This happens because every node saves the nearest pending request from both left and right directions.

Figure 3, shows a bar chart of messages required per CS execution. As we can see, as the load increases on the system, number of messages gets reduced in both algorithms. In Neamatollahi's Algorithm, node forwards the request message only when it does not have request of any other node in the queue. Therefore, as load increases, the nodes request for CS frequently, so most of the time nodes have requested message of other nodes in its request queue.Similarly, in ITDME algorithm, if a node's left and right neighbour have already requested for CS, then they do not forward the node's request message further because for another nodes, its neighbour will be the nearest node. Hence, as load increases from low to high, messages per CS get reduced.

Table 2. Average number of messages required per CS execution for 25 nodes

Request/sec	Neamatollahi's Algorithm	ITDME Algorithm
0.1	44.78021053	48.40968421
0.2	22.03957895	24.36
0.3	14.98568421	17.73726316
0.4	10.36547368	13.92673684
0.5	8.210526316	11.67915789
0.6	6.789473684	10.55326316
0.7	6.050105263	9.857263158
0.8	5.176421053	8.962947368
0.9	4.602105263	8.223578947
1.0	3.787789474	8.282526316

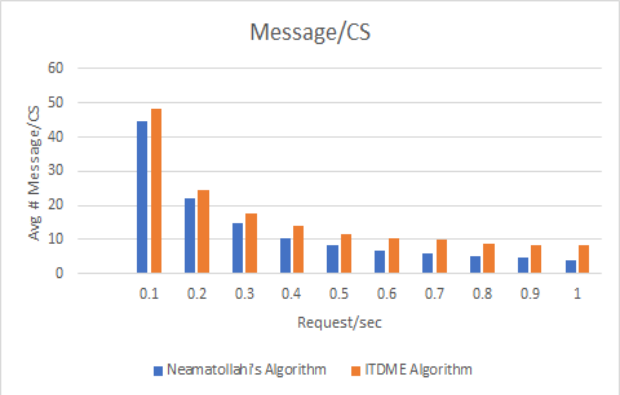

Fig. 3. Average number of messages required per CS execution for 25 nodes

In table 3, average time delay for executing CS is shown. In Neamatollahi's Algorithm, despite of position of requesting node, the token will start from right neighbour's and complete its horizontal cycle before moving in a downward direction. This is the main reason of high time delay before executing CS.

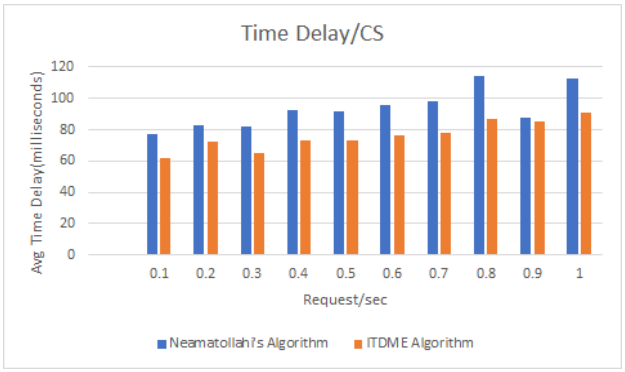

Fig. 4. Average Time Delay per CS for 25 nodes (in ms)

Table 3. Average Time Delay per CS for 25 nodes (in ms)

Request/sec	Neamatollahi's Algorithm	ITDME Algorithm
0.1	77.51949053	61.78188295
0.2	82.85520337	72.65979453
0.3	82.40457011	65.37628547
0.4	92.77552168	72.90117642
0.5	91.49594737	73.20732884
0.6	95.82246526	76.51554611
0.7	98.38707411	78.02325221
0.8	114.1647587	86.95598063
0.9	87.44365179	85.44419916
1.0	112.9700632	91.28418316

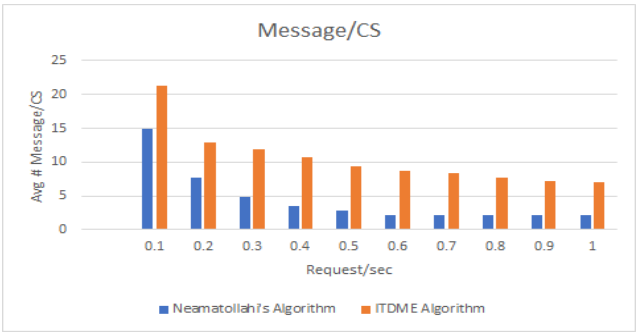

Fig. 5. Average number of messages per CS execution for 64 nodes

According to the results shown in Figure 4, ITDME algorithm takes less time delay compared to Neamatollahi's algorithm because it chooses the next requesting node according to nearest request present in its ReqArr.

If the node has a pending request in both 0th and 2nd positions then it chooses the nearest between those two. This reduces the extra movement of the token, which ultimately helps reduce time delay.

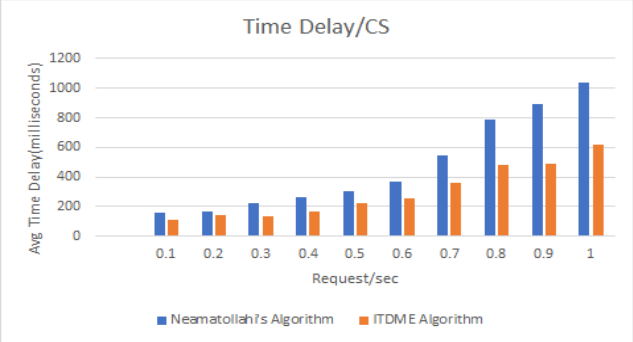

Fig. 6. Average Time Delay per CS for 64 nodes (in ms)

Effect of Arrival Rate on a 64-Node System:

As In Figure 5 and Figure 6, we have shown the effect of arrival rate on average message per CS and time delay on a 64 node system.

Effect of Varying Number of Nodes:

Table 4 shows the average number of messages per CS required by both algorithms. It is the average of all arrival rates from 0.1 to 1.0. As the number of nodes increases from 25 to 100, the number of messages gets decreased, and after a certain level, it becomes almost constant in both algorithms.

Table 4. Average number of messages per CS execution for different nodes

Number of Nodes	Neamatollahi's Algorithm	ITDME Algorithm
25	12.67873684	16.19924211
36	9.764649123	12.66330409
49	5.761976369	11.26234157
64	4.452697368	10.48723684
81	4.48245614	9.815126706
100	3.688589474	9.580336842

In Figure 8, time delay for executing CS is shown. As number of nodes increases, the time delay in both algorithms increases. ITDME has less time delay as compared to Neamatollahi's algorithm because of intelligent movement of the token.

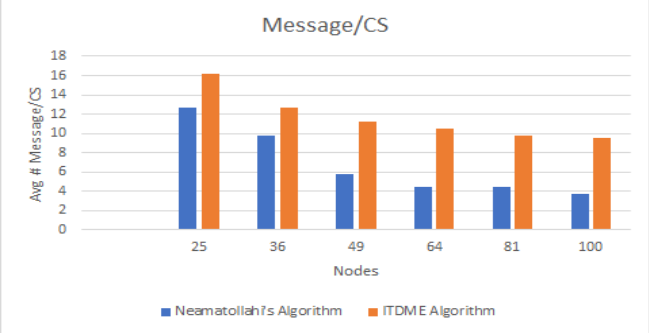

Fig. 7. Average number of messages per CS execution for different nodes

Table 5. Average time delay per CS execution for different nodes

Number of Nodes	Neamatollahi's Algorithm	ITDME Algorithm
25	93.58387461	76.41496295
36	112.7201494	104.8959949
49	277.5155715	162.5997615
64	477.4028808	299.9316277
81	484.9101446	397.4457022
100	736.3363589	576.6257086

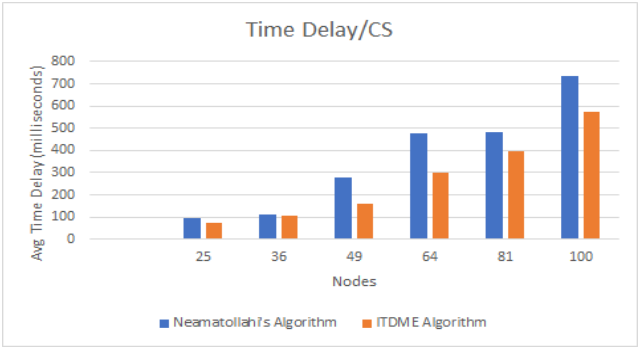

Figure 8. Average Time Delay per CS execution for different nodes in milliseconds

V. CONCLUSION

This paper presents an improved distributed algorithm for mutual exclusion problem in a distributed scenario. The two dimensional torus is used as a logical structure. CS requests are forwarded in a horizontal ring while token moved in vertical ring. We proved that ITDME satisfy safety and liveness property. By moving token in a greedy manner we can reduce the average waiting time. The limitation of this algorithm is that it requires a high number of messages to execute the CS under light load. It happens because the token continuously moves in the system. However, it works better with high load. The present algorithm assumes that $N = d^2$, where d is an integer and N is the total number of nodes. So for future work, we can try to remove this constraint by using torus topology having m rows and n columns. This can a ect the total number of messages required to execute the CS. In present algorithm, fault tolerance mechanism can be added.

REFERENCES

1. J. Wu, *Distributed System Design*. CRC press, 2017.
2. S. Ghosh, *Distributed systems: an algorithmic approach*. Chapman and Hall/CRC, 2014.
3. K.-H. N. Bui and J. J. Jung, "Internet of agents framework for connected vehicles: A case study on distributed traffic control system,"*Journal of Parallel and Distributed Computing*, vol. 116, pp. 89-95, 2018.
4. J. Lim, Y. S. Jeong, D.-S. Park, and H. Lee, "An efficient distributed mutual exclusion algorithm for intersection traffic control,"*The Journal of Supercomputing*, vol. 74, no. 3, pp. 1090-1107, 2018.
5. P. Neamatollahi, Y. Sedaghat, and M. Naghibzadeh,"A simple token-based algorithm for the mutualexclusion problem in distributed systems," *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3861–3878, Sep2017.
6. D. Seo, S. Kim, and G. Song, "Mutual exclusion method in client-side aggregation of cloud storage," IEEE Transactions on Consumer Electronics, vol. 63, no. 2, pp. 185-190, 2017.

7. M. Bienkowski, M. Klonowski, M. Korzeniowski, and D. R. Kowalski, "Randomized mutual exclusion on a multiple access channel," *Distributed Computing*, vol. 29, no. 5, pp. 341-359, 2016.
8. G. Ricart and A. K. Agrawala, "An optimal algorithmfor mutual exclusion in computer networks," *Commun. ACM*, vol. 24, no. 1, pp. 9–17, Jan. 1981.
9. M. Singhal, "A heuristically-aided algorithm for mutualexclusion in distributed systems," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 651–662, May 1989.
10. E. W. Dijkstra, "Solution of a problem in concurrentprogramming control," *Commun. ACM*, vol. 8, no. 9,pp. 569–, Sep. 1965.
11. N. Yadav, S. Yadav, and S. Mandiratta, "Article:A review of various mutual exclusion algorithmsin distributed environment," *International Journal of Computer Applications*, vol. 129, no. 14, pp. 11–16, November 2015.
12. M. Raynal, "A simple taxonomy for distributedmutual exclusion algorithms," *SIGOPS Oper. Syst. Rev.*,vol. 25, no. 2, pp. 47–50, Apr. 1991.
13. L. A. Rodrigues, E. P. Duarte Jr, and L. Arantes, "A distributed k-mutual exclusion algorithm based on autonomic spanning trees," *Journal of Parallel and Distributed Computing*, vol. 115, pp. 41-55, 2018.
14. S.-H. Park and S.-H. Lee, "Quorum-based mutual exclusion in asynchronous distributed systems with unreliable failure detectors," *TheJournal of Supercomputing*, vol. 67, no. 2, pp. 469-484, 2014.
15. L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 1–11, Jan. 1987.
16. P. K. Srimani and R. L. Reddy, "Another distributedalgorithm for multiple entries to a critical section,"*Information Processing Letters*, vol. 41, no. 1, pp. 51– 57, 1992.
17. P. Chaudhuri and T. Edward, "An O($\sqrt{N}$) distributedmutual exclusion algorithm using queue migration,"*Journal of Universal ComputerScience*, vol. 12, no. 2,pp. 140–159, feb 2006.
18. B. Sharma, R. S. Bhatia, and A. K. Singh, "A token based protocol for mutual exclusion in mobile ad hoc networks," *Journal of Information Processing Systems*, vol. 10, no. 1, pp. 36-54, 2014.
19. H. Taheri, P. Neamatollahi, and M. Naghibzadeh,"A hybrid token-based distributed mutual exclusionalgorithm using wraparound two-dimensional arraylogical topology," *Information Processing Letters*, vol.111, no. 17, pp. 841 – 847, 2011.
20. S. A. Tamhane and M. Kumar, "A token based distributed algorithm for supporting mutual exclusion in opportunistic networks," *Pervasive and Mobile Computing*, vol. 8, no. 5, pp. 795-809, 2012.
21. I. Suzuki and T. Kasami, "A distributed mutualexclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 3,no. 4, pp. 344–349, Nov. 1985.
22. M. Mizuno, M. L. Neilsen, and R. Rao, "A tokenbased distributed mutual exclusion algorithm basedon quorum agreements," *in Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 361–368, May 1991.
23. Y. I. Chang, M. Singhal, and M. T. Liu, "Adynamic token-based distributed mutual exclusionalgorithm," *in Proceedings of the Tenth Annual International Phoenix Conference on Computers and Communications*, pp. 240–246, Mar 1991.
24. Y. Yan, X. Zhang, and H. Yang, "A fast token-chasingmutual exclusion algorithm in arbitrary networktopologies," *J. Parallel Distrib. Comput.*, vol. 35, no. 2,pp. 156–172, Jun. 1996.
25. P. Saxena and S. Gupta, "A token-based delay optimalalgorithm for mutual exclusion in distributed systems,"*Computer Standards & Interfaces*, vol. 21, no. 1, pp. 33–50, 1999.
26. Y.-I. Chang, "A simulation study on distributed mutualexclusion," *J. Parallel Distrib. Comput.*, vol. 33, no. 2,pp. 107–121, Mar. 1996.

AUTHORS PROFILE

Prashant Kumar received his B. Tech degree in Computer Science and Engineering from SRMCEM Lucknow (U.P.), India in 2013 and M. Tech in Software Engineering from Motilal Nehru National Institute of Technology Allahabad (U.P.), India in 2018. Presently he is working in Veritas Technologies LLC.His main research interest lies in Distributed Systems and Machine Learning.

Naveen Kumar Gupta received his B. Tech degree in Computer Science and Engineering from Inderprastha Engineering College Ghaziabad (U.P.), India in 2011 and M. Tech in Information Technology from Madan Mohan Malaviya University of Technology Gorakhpur (U.P.), India in 2014. Presently he is pursuing PhD from Motilal Nehru National Institute of Technology Allahabad (U.P.), India since July, 2015. He also worked as Assistant Professor at SR Group of Institutions Jhansi (U.P.), India from August-2014 to July-2015. His main research interest lies in Wireless Sensor Networks and Ad hoc Routing protocols.

Rama Shankar Yadav is currently a professor at Motilal Nehru National Institute of Technology, Allahabad, India. He received his Ph.D. degree from the Indian Institute of Technology (IIT), M.S. degree from Birla Institute of Technology and Science (BITS) Pilani, and B. Tech. degree from the Institute of Engineering and Technology (I.E.T.), Lucknow, India. Dr. Yadav has extensive research and academic experience. He has worked in leading institutions such as Govind Ballabh Pant Engineering College (GBPEC), Pauri, Garhwal, and Birla Technical Training Institute (BTTI), Pilani. He has authored more than 70 research papers in national/international conferences, refereed journals, and book chapters. Dr. Yadav's areas of interest are real time systems, embedded systems, fault-tolerant systems, energy aware scheduling, network survivability, computer architecture, distributed computing, and cryptography.

Rajendra Kumar Nagaria received B.Tech and M.Tech degree in electronics engineering from KNIT Sultanpur, India in 1988 and 1996 respectively and the Ph.D.(Engg.) degree from Jadhavpur University Kolkata, India in 2004. He is currntly working as a full Professor in the department of Electronics & Communication Engineering, Motilal Nehru National Institute of Technology Allahabad, India since January 2009. He has 28 years of teaching and research experience and contributed more than eighty research articles. He is fellow of professional bodies like The Institution of Engineers (India), Indian Society for Technical Education and member of IEEE. His main research interest includes Analog/Mixed-mode circuits and Low power VLSI circuits and systems.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication (BEIESP)