International Journal of Electrical, Electronics and Computer Engineering **2**(2): 160-169(2013)

ISSN No. (Online): 2277-2626

A Modified Distributed Approach for Mutual Exclusion with Increased Memory and CPU Utilization

Rahul Singh*, Sunita Gond** and Deshraj Ahirwar***

**Department of Information Technology, BUIT BU Bhopal, (MP)*

***Department of MCA, BUIT BU Bhopal, (MP)*

****Department of Computer Science and Engineering, UIT RGPV Bhopal, (MP)*

(Received 05 November, 2013 Accepted 07 December, 2013)

ABSTRACT: A distributed system consists of a collection of geographically dispersed autonomous sites connected by a communication network. The sites have no shared memory and communicate with one another by passing messages. To achieve mutual exclusion concurrent access to a shared resource or the Critical Section (CS) must be synchronized such that at any time only one process can access the CS. Over the past decade, many algorithms to achieve mutual exclusion in distributed systems have been proposed. Those algorithms have succeeded in minimizing either message traffic or time delay. However, there is no single algorithm that can increase CPU and memory utilization at the same time. To increase CPU and memory utilization at the same time, we must look for a modified distributed approach to mutual exclusion in which one special message reply is added to ensure exact time for waiting to enter in critical section. The process which is waiting for entering the critical section for a long time while being in the main memory, we can make the process move to secondary memory for some time, So that small processes which were waiting for other resources but could not enter in main memory due to lack of space in the main memory can be executed.

I. INTRODUCTION

In distributed system cooperating processes can effect or be affected by other processes in the system. Cooperating processes either share address space (that is both code and data) or be allowed to share data only through files or messages. The former can be implemented through multithreading. Concurrent access to shared data may result in data inconsistency [1]. The critical section problem is available almost in every text book of operating systems. Consider there are n numbers of processes which are competing to use some shared data. Each process has a code segment, called critical section, in which it can access and manipulate shared data. If concurrent processes accessing the shared common resource are not synchronized such that only one process can access this shared resource, then it will lead to integrity violations. CS Problem is to guarantee that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

In distributed systems, cooperating processes share both local and remote resources. Chance is very high that multiple processes make simultaneous requests to the same resource.

If the resource requires mutually exclusive access (critical section – CS), then some regulation is needed to access it for ensuring synchronized access of the resource so that only one process could use the resource at a given time. This is the distributed mutual exclusion problem [2]. of the resource so that only one process could use the resource at a given time. If the resource requires mutually exclusive access (critical section – CS), then some regulation is needed to access it for ensuring synchronized access. This is the distributed mutual exclusion problem [2].

There are several definitions and view points on what distributed systems are. Coulouris defines a distributed system as “a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing” [3]; and Tanenbaum defines it as “A collection of independent computers that appear to the users of the system as a single computer” [4]. Leslie Lamport – a famous researcher on timing, message ordering, and clock synchronization in distributed systems once said that “A distributed system is one on which I cannot get any work done because some machine never heard of has crashed” reflecting on the huge number of challenges faced by distributed system designers.

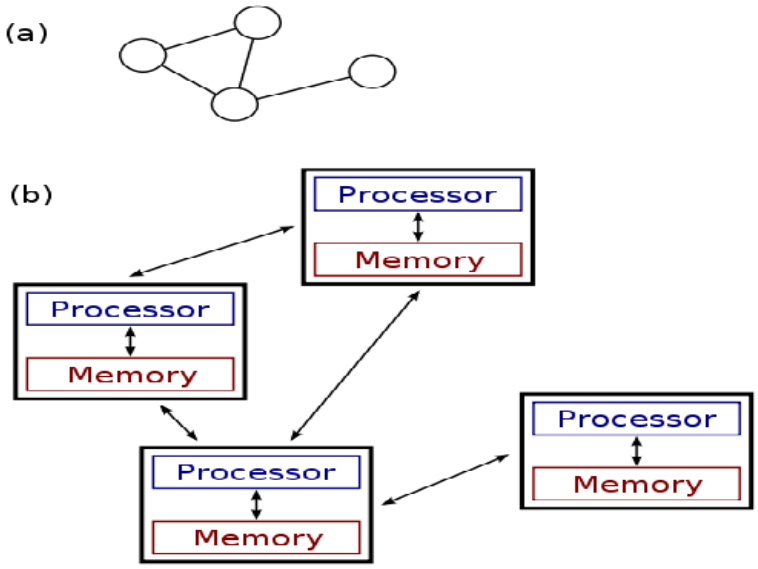


Fig. 1. Distributed system.

Despite these challenges, the benefits of distributed systems and applications are many, making it worthwhile to pursue [5].

In most of the distributed systems it is very common that, resources are being shared among various processes, with the condition that a single resource can be allocated to a single process at a time. Therefore, mutual exclusion is a fundamental problem in any distributed computing system. So, the goal is to find a solution that will synchronize the access among shared resources in order to maintain their consistency and integrity [6].

II. MUTUAL EXCLUSION

Systems involving multiple processes are often most easily programmed using critical regions [7].

When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

A. Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address).

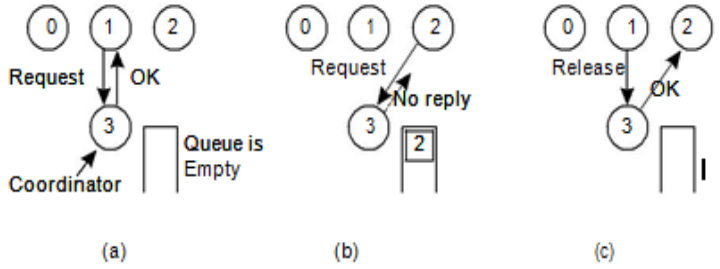


Fig. 2. (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig. 2 (a). When the reply arrives, the requesting process enters the critical region.

A. Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system
is illustrated in Fig. 3.
Here we have a bus network, as shown in Fig. 3(a), (e.g., Ethernet), with no inherent ordering of the processes.

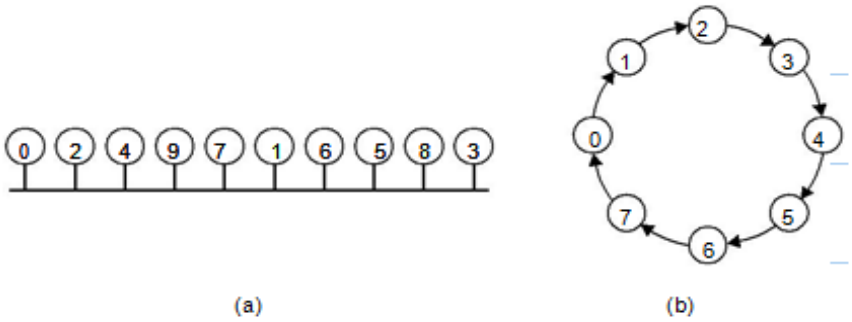


Fig. 3 (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig. 3(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process *k* to process *k* 1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

B. Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method. Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first.
The algorithm works as follows.
When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available can be used instead of individual messages. When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:
(i) If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
(ii) If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
(iii) If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message.

If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing. After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region.

When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue. Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 4(a).

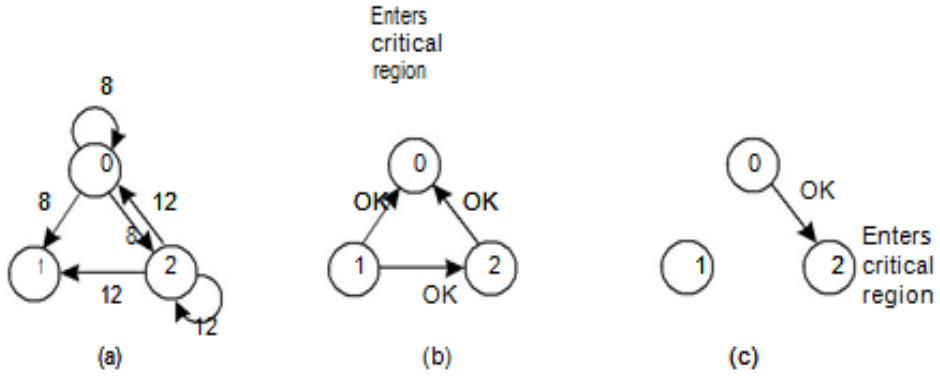


Fig. 4 (a). Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 4(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig. 4(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

III. RELATED WORK

Mi-Sook Kim and Reda A [8] says that In shared memory environment, processes in a parallel structure communicate with one another via common variables. Since no two processes should access shared variables simultaneously therefore they should be placed in the critical section to ensure mutually exclusive access. One way to divide up the work in a shared memory system is the fork-join structure where the FORK statement spawns several processes and JOIN is used to synchronize the termination of processes [9]. The portion of program between the FORK and JOIN is called the parallel structure [10,11].

This paper [8] says that a scheduling algorithm using a partitioning tool can be used as a way towards improving the results produced by the prior scheduling approach. The problem is how to order processes competing to traverse the critical section with the task of minimizing the time spent to execute these processes. The simulation showed that this algorithm produces even better outcome. However we did not consider any overhead from allowing preemptive access to the critical section in this paper Previous heuristic algorithm scheduled these processes without allowing preemption in accessing the critical section.

Xiao Peng1[10] presents a kernel level distributed interprocess communication system model with support for distributed process synchronization and communication. With the development of computer networks, especially the development of the Internet, distributed systems have been applied to all kinds of fields. It has played an important role in people's daily life. It is necessary for a distributed system to offer a powerful and flexible inter process communication function, and to effectively release and get information in a wide area of computing environment. Hence, inter process communication of the distributed computing systems becomes an important question for discussion.

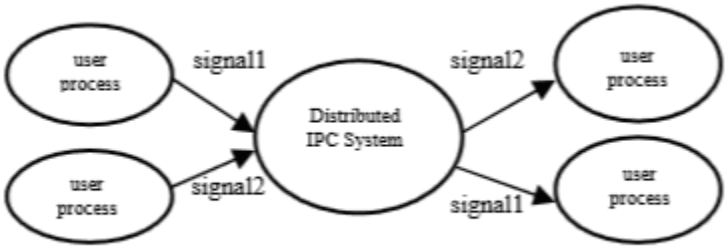


Fig. 5. Distributed IPC.

Xiao Peng1 [12] says that, The Distributed Interprocess Communication System is also based on pure TCP/IP protocol and C/S model to provide service to user processes, as shown in Figure 5.
Sandipan Basu [13] presents an algorithm for achieving mutual exclusion in Distributed System. Proposed algorithm does not allow the circulation of the token along the ring, when there is no need (i.e. when no process wants to enter in its critical section). In the already existing algorithm, there are few problems, which, if occur during process execution, then the distributed system will not be able to ensure mutual exclusion among the processes and consequently unwanted situations may arise. Loss of a token in the ring can easily be detected, and regeneration of token can be done easily in this algorithm. And process crash and recovery of crashed process can easily be managed using this algorithm.

IV. PROPOSED WORK

In distributed approach a change has been made as a part of my research to increase the memory and CPU utilization by temporarily swapping out those processes which have been waiting for a long time and entering small processes in the main memory. Distributed approach is used for implementing mutual exclusion by sending messages to all other active processes. It only implements mutual exclusion. It requires reply messages from all other processes to ensure that the resource is available to it.

V. ALGORITHM

Step 1. There are n processes P1 to Pn.

Step 2. If Pi sends request message to all other n-1 processes with its own process id, the name of the critical section that the process wants to enter and a unique timestamp generated by the process for the request message.

Step 3. IF process Pj(other than Pi) on receiving a request message, neither is in the critical section nor is waiting for its chance to enter the critical section go to step 7.

Step 4. Else If process Pj(other than Pi) on receiving a request message, is itself currently executing in the critical section

```
Do
{
        Process Pj queues the received request message
        Go to step 8
}
```

Step 5. Else If process Pj(other than Pi) on receiving a request message, is currently not executing in the critical section but is waiting for its chance to enter the critical section.

```
Do
{
        Pj compares the timestamp in the received request message with the timestamp in its own request message that it has sent to other processes.
        If the timestamp of the received request message is lower
        Do
        {
                Go to step 7
        }
        Else
        Do
        {
                Go to step 6
```

}

Step 6. Process Pj queues the received request message.

Est = Est(old) + own burst time

Send pre reply message with Est value to Pi.

Go to step 9

Step 7. Process Pj immediately sends a reply message to the Pi(sender).

Go to step 9

Step 8. After Pi finishes executing in the critical section, it sends reply message to all processes in its queue and deletes them from its queue.

Go to step 12.

Step 9. If Pi receiving special message from Pj with Pj's Est value

Do

{

If Est>limit

Do

{

Send special reply msg with its own Est value to that Process, from where Pi got highest Est value.

Swap out Pi for (Est - (swap out time + swap in time)) time to secondary memory.

Go to step 10

}

Else

Do

{

Go to 11

}

}

Go to 11

Step 10. After (Est - (swap out time + swap in time)) time, Process Pi again swapin in main memory.

Go to step 11

Step 11. If Pi got reply message from every processes

Do

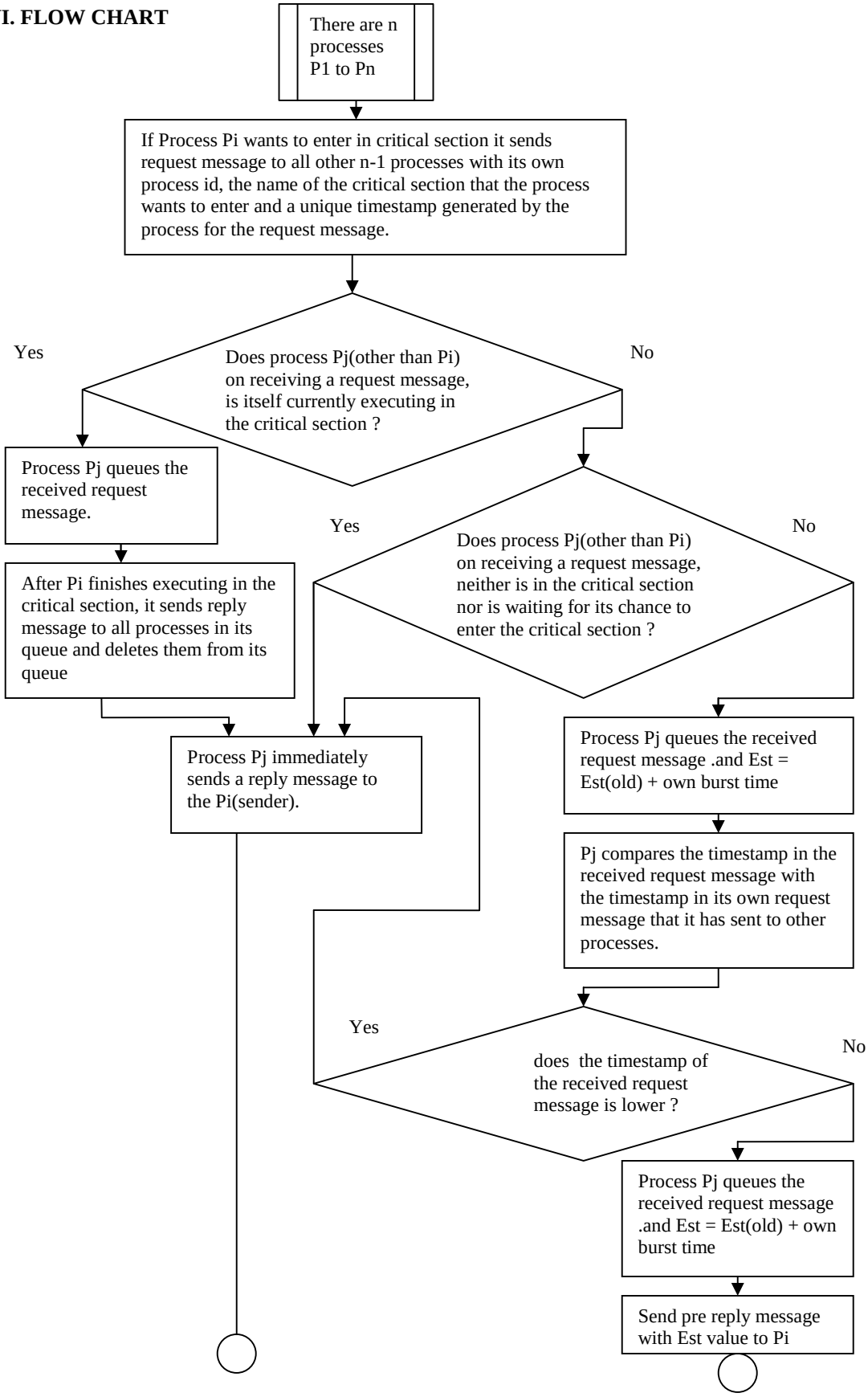
{

Go to step 8

}

Step 12. Exit.

VI. FLOW CHART



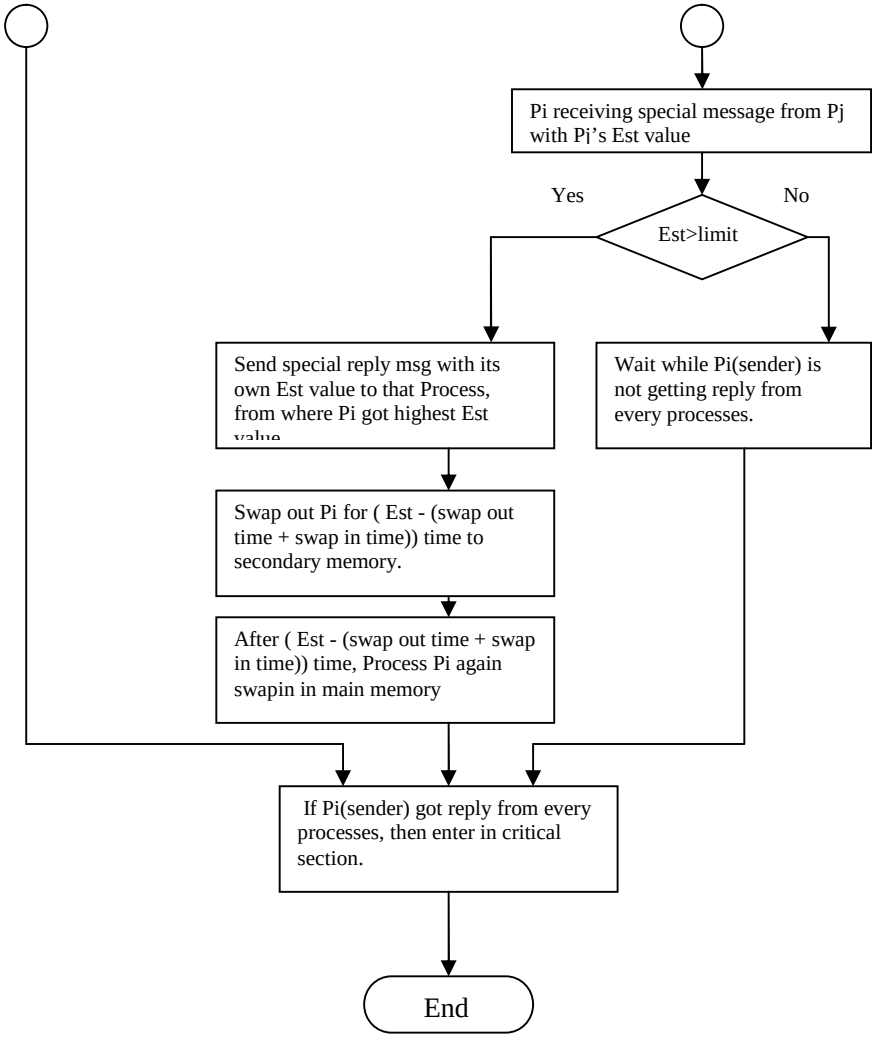


Fig. 6. Flow chart of a Modified Distributed Approach For Mutual Exclusion With Increased Memory And CPU Utilization.

VII. RESULTS

Table 1.

Process ID	Arrival time	Burst time
P1	3	5
P2	4	2
P3	2	4
P4	4	1
P5	8	2

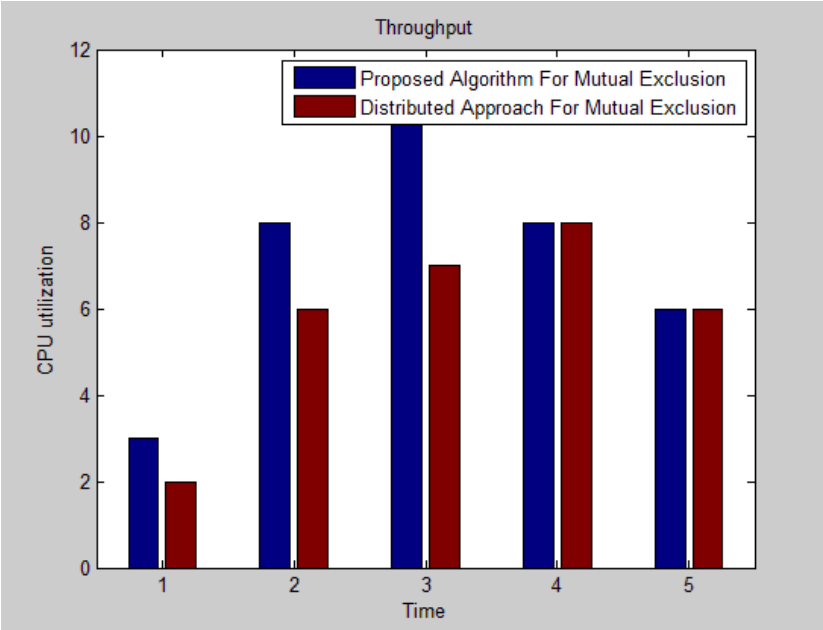


Fig. 7. Comparison of throughput of distributed algorithm and proposed algorithm for given values in table 1.

Table 2.

Process ID	Arrival time	Burst time
P1	9	8
P2	4	4
P3	1	7
P4	8	2
P5	3	7

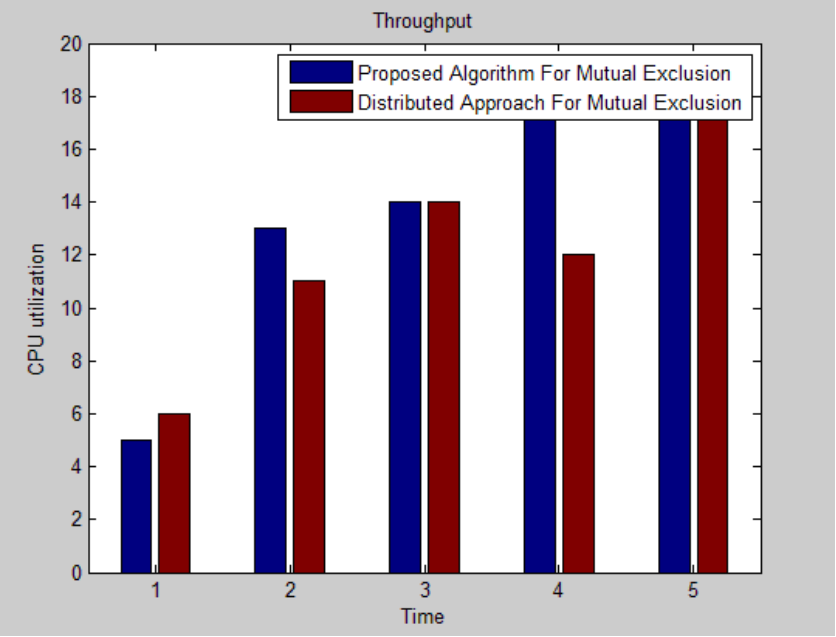


Fig. 8. Comparison of throughput of distributed algorithm and proposed algorithm for given values in table 2.

VIII. CONCLUSION

Distributed approach can be utilized for implementing mutual exclusion by sending messages to all active processes. It implements mutual exclusion. It requires response from all other processes to ensure that the resource is available to it. This is where we make it sure that the waiting process is provided with a reply from the process which holds the resource, this in turn, makes it sure that the memory held by the waiting processes is freed and the burst time of all the processes is known to every other process which is in the queue and thus the solution is pertainable and is clearly better than the usual approach which was being followed till the date. Though there is a constraint regarding the time taken in sending the message which needs to be taken care of. Resource allocation techniques which primarily implements the mutual exclusion concept which makes it sure that the editing by multiple processes of a chunk of data must not be executed simultaneously. Therefore the utilization of input output devices must be handled by one process at a time. Here the time management can be proved to be critical and hence special care is required to have it managed in order to keep the execution of the system smooth and utter.

IX. FUTURE WORK

Future holds the key. These investigations have identified some interesting directions :

1. Reducing the complexity of the algorithm by making it simpler despite having lots of processes and resources.
2. Time management can be made better via research in future.
3. Resource allocation can be made simpler by adding a newer algorithm.

REFERENCES

[1]. Md. Abdur Razzaque Choong Seon Hong, 2008, "Multi-Token Distributed Mutual Exclusion Algorithm," in 22nd *IEEE International Conference on Advanced Information Networking and Applications*, 1550-445X/08, AINA, pp. 963–970.

[2]. E.W. Dijkstra, 1965, Solution of a Problem in Concurrent Programming Control, Communication ACM, vol. 8, no. 9, Sept.

[3]. G. Couloris, J. Dollimore, and T. Kinberg, 2001, Distributed Systems - Concepts and Design, 4th Edition, Addison-Wesley, Pearson Education, UK.

[4]. A. Tanenbaum and M. Van Steen, 2002, Distributed Systems: Principles and Paradigms, Prentice Hall, Pearson Education, USA.

[5]. Krishna Nadiminti, Marcos Dias de Assunção, and Rajkumar Buyya, 2013, Distributed Systems and Recent Innovations: Challenges and Benefits Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering ,The University of Melbourne, Australia

[6]. Token Ring Algorithm To Achieve Mutual Exclusion In Distributed System – A Centralized Approach Sandipan Basu Post Graduate Department of Computer Science, St. Xavier's College, University of Calcutta Kolkata-700016, INDIA.

[7]. Andrew S. Tanenbaum, Maarten van Steen, Distributed Systems: Principles and Paradigms

[8] M.-S. Kim, R.A. Ammar, 1998, "Using Preemptive Access to the Critical Section in Shared Memory Environment to Minimize the Execution Time of the Fork-Join Structure," iscc, pp.559, *Third IEEE Symposium on Computers & Communications.*

[9]. Alan H. Karp, May 1987 "Programming for Parallelism", IEEE Com- puter, pp. 43-57.

[10]. Reda A. Ammar, A.I. El-Desouky, T.A. Fergany, and M.M. Hefeeda. Sep., 1996 "Heuristic scheduling algorithms to access the crit- ical section in Shared Memory Environment," Proceedings of the ISCA conference on Parallel and Distributed Computing Systems, Dijon, France, pp. 244-247.

[11]. T.G. Lewis and H. El-Rewini, 1992, "Introduction to parallel com- puting," Prentice–Hall, Inc..

[12]. Xiao Peng, Li Yuanyuan, 2009, A Model of Distributed Interprocess Communication System, *IEEE DOI 10.1109/WKDD.2009.37*

[13]. Sandipan Basu, January 2011, Token Ring Algorithm To Achieve Mutual Exclusion In Distributed System – A Centralized Approach, *IJCSI International Journal of Computer Science* , Vol. **8**, Issue 1, ISSN (Online): 1694-0814 www.IJCSI.org.