Bitlis Eren Univ J Sci & Technol
6 (1), 16-21, 2016

E - ISSN 2146-7706

Research Article

Petri Net Model for Ricart and Agrawala's Mutual Exclusion Algorithm

Vassilya Uzun [1] *

[1] *Alanya HEP University, Department of Graphics Design, Alanya - Turkey*
* *Corresponding author: vassilya.abdulova@gmail.com*

Abstract

Issues involved in mutual exclusion and background of mutual exclusion are discussed. A Ricart and Agrawala mutual exclusion algorithm is investigated. Simulation model of the system based on Petri Nets is described. Simulation results are presented and evaluated

Keywords: Mutual exclusion, Petri nets, simulation.

1. Introduction

Mutual exclusion algorithms, which are frequently named mutex, are utilized in concurrent programming in order to prevent critical sections' concurrent utilization of non-shareable resources. These resources can be exemplified with fine-grained flags, counters, queues, and other data utilized in order to communicate between code running when an interrupt is being serviced, and the code running during the rest of time. If there is no special attention, an interrupt can easily emerge between 2 non-interrupt code instructions, and this problem can be defined to be acute for this reason. When the protection of the critical section isn't ensured, this situation may lead to serious failures. The routine method utilized for achieving the mutual exclusion is to eliminate the interrupts for the smallest possible number of instructions. These instructions will eliminate any corruption within the shared data structure named "critical region". By implementing this method, the interrupt code will not run in the critical region (Khanna, 2014), (Peterson, 1981).

Many processors utilize the shared memory in any computer, and the reason of using an indivisible test-and-set of a flag is to wait until the other computer clears the flag. A test-and-set CPU instruction ensures that it will test and change the content in determined memory locations. In multi-processor settings, this is used to implement semaphores. In uniprocessor settings, it can eliminate the interrupts before the access to semaphore. On the other hand, in multiprocessor settings, synchronously disabling the interrupts in all of the processors is not possible, as well as it is not desired. Even if it is succeeded, it is possible that two or more processors would try to attempt to access the same semaphore's memory concurrently. The test-and-set instruction allows any processor to atomically test and modify a memory location, and it also eliminates such multiple processor collisions. The test-and-set executes all the operations without allocating the memory bus to the other processor. When the code stops running in critical region, it will clear the flag. This process is named "spin lock" or "busy-wait." The spinlock is the lock where the thread keeps checking in a loop (named "spins") until the lock becomes available. This process is also named "busy waiting" since the thread keeps existing but it doesn't execute a useful task. Once acquired, spinlocks will be held until they are released or the thread blocks (goes to sleep) (Khanna, 2014), (Peterson, 1981).

Mutexes have some forms having various side-effects. For example, classic semaphores allow deadlocks. In these deadlocks, while a process gets a semaphore, another process also gets a semaphore. They both wait for the other semaphore to be released. Another side-effect includes "starvation", in which an essential process does not run enough, and "priority inversion" in which a higher priority task waits for a lower-priority task, and "high latency" in which response to interrupts is not prompt (Khanna, 2014), (Peterson, 1981).

1.1 Issues Involved in Mutual Exclusion

While there is an entry section at the beginning of the code's critical section, while the section has an exit section at its ending. The lock of that section is arranged by these sections. About the distributed and concurrent algorithms, there are two kinds of properties; the safety property and the progress property. As it can be understood from its name, the safety property eliminated the possibility of any bad thing. Since the static status of any setting at certain time point is taken, it is obviously easy to talk about proving them in algorithms. But, the other property, the progress property, ensures the occurrence of good things. It is much more complicated to prove this, because the factor "time" must be taken into account. An important part of programming is to prove the accurateness of the program with concurrent algorithms since the values of the variables can vary depending on the statements even if there is no change done by the process we are designing, and it violates our intuitive notion of program flow. The satisfaction of the properties mentioned above can be proved only via formal proof techniques (Khanna, 2014), (Peterson, 1981). With the problem of mutual exclusion, one safety property is that of mutual exclusion; that is, not more than one process should have its program counter (PC) in the critical code at the same time. In order to validate this, it is enough to demonstrate that one process's PC cannot leave the entry section while that of other process is between the entry and exit sections. Freedom from any deadlock is another desirable safety property. This

Received 27 January 2016
Available online June 2016

feature ensures that all processes won't be stuck in critical section (Khanna, 2014), (Peterson, 1981).

In mutex algorithms, there are 2 desirable progress properties. First one is the freedom from livelock. This feature can be explained with sentence "If some process wants to enter the critical section, then some process will eventually enter the critical section". The other progress property, freedom from starvation, is stronger than the first and is phrased as, "If any process wants to enter into the critical section, then that process will do it sooner or later". The second one indicates the other one, but it is much harder to guarantee it. Please note that the freedom from livelock also indicates the freedom from deadlock (Khanna, 2014), (Peterson, 1981).

In literature, we frequently see that the terms process and processor may be used interchangeably. This is because it is generally thought that only one kernel-level process is running on each processor. But it is also possible to handle multiple processes on a single processor through a traditional way.

1.2 History of Mutual Exclusion

A lot of research has been done in the field of mutual exclusion. In year 1962, T. Dekker has proposed the problem of multiprocessor mutex firstly. But until 1965, there was no accurate solution for 2+ processes. In that year, Edsger Dijkstra has written his one-page article (Dijkstra, 1965) which provided a working but very complicated solution for this problem. In his proposal, the mutual exclusion, freedom from deadlock, and freedom from livelock were guaranteed, but the proposal also allowed the possibility of one process being forever stuck in the entry section while other ones are allowed into the critical section. Right after his paper, Donald Knuth proposed an equally complicated solution (Knuth, 1966) guaranteeing the freedom from starvation.

There was a fault in both of the mentioned algorithms. In that fault, if a single processor fails at any point within the entry section, the entire system might become blocked indefinitely. Moreover, both of two solution proposals were assuming atomic reads and writes to memory. This means that they assumed that the reading and writing actions to single memory locations will never overlap thanks to the hardware. This assumption doesn't always work well, and we should be ready for a read overlapping a write. This situation will lead to invalid partial results. If the writes overlap, this situation would lead to writing invalid or partial data.

8 year after the writing of Knuth, Leslie Lamport proposed a solution (Lamport, 1974) which is not only simpler that the first two, but which both allows any processor to halt anywhere in the entry section and allows a read to return any arbitrary value if it overlaps a write. In Lamport's algorithm, there was an additional property of "first come-first served". This property has not been offered by Dijkstra's and Knuth's algorithms.

In year 1981, Gary Peterson has designed his algorithm (Peterson, 1981) for processes just 7 years after the article of Knuth, besides the algorithm for more than 2 processes. The two-process algorithm was very simple. For this reason he has through that there was no need for any validation of accuracy:

```
/* Entry section for P1 */          /* Entry section for P2 */
    Q1 := True;                         Q2 := True;
    TURN := 1;                          TURN := 2;
    wait while Q2 and TURN := 1;        wait while Q1 and TURN := 2;
/* Exit section for P1 */           /* Exit section for P2 */
    Q1 := False;                        Q2 := False;
```

In both of algorithms, the processes wait until they will be able to enter into the critical section.

When it comes to multiprocessor systems, re-assessment of the solutions utilized in the uniprocessor systems is of significant importance. For example; the mutex is a rational solution in uniprocessor systems since only 1 process can be executed at any one time, and then the mutex really just specifies their order of scheduling. On the other hand, when it comes to multiprocessor systems, the mutex is not a cost-efficient operation. The reason is not the overhead involved, is also that one processor will sit idle while another is able to access to the data.

Leslie Lamport introduced an algorithm (Lamport, 1977) that will always allow a single writer to write without any blockage. The core point was to allow the writer to write at any time, and the reader to read the data over and over until it is clear that it has a valid copy. It is important for the writer to write data version numbers before and after the update, for the reader to read and compare them.

If the mutex is the only option, then, especially when it comes to largely distributed systems, we will maybe need to face with the large delays in the entry section. Among the designers of operating systems, the thought that most of the processes entering into the entry section are allowed to enter the critical section without any delay has become a belief. For this reason, many algorithms trying to reduce the O(N) access time of the initial mutual exclusion algorithms have been developed. Leslie Lamport has introduced his mutual exclusion algorithm leading once again (Lamport, 1987) and requiring only 7 memory accesses in the case of no contention.

As it is known, majority of the distributed systems have been adjusted so that each processor has rapid access to its local memory and slow access to other processors' ones. Generally, all of the processors utilize the same interconnect network, and the communication between the processors, even ones not attempting to enter into any critical section, is significantly affected by the remote spin-locking. The expectations focused on the minimization of the Attention turned to minimizing the remote memory accesses, and then Mellor-Crummey and Scott has designed a new algorithm (Mellor-Crummey, 1991) which had O(1) remote references.

One of the core parts of the distributed and concurrent computing is the access to the common-used data. The algorithms to mediate the concurrent accesses are the core features of the distributed operating systems. The algorithm of choice is based on the hardware support and also the hardware configuration. If the durability is expected, then the wait-free designs will ensure that unexpected processor dysfunctions won't affect the other parts of the systems.

The rest of the paper is structured as follows. In Section 2, Ricart and Agrawala's mutual algorithm is described. The simulation model based on Petri Nets is discussed. Section 3 presents and evaluates the results of the simulation.

2. Ricart and Agrawala's Algorithm

This paper is related to Ricart and Agrawala's Mutual Exclusion Algorithm (Ricart, Agrawala, 1981) and the

Petri net (Murata, 1989) implementation of this model using Winsim (Kostin, Ilushechkina, 2005) simulation tool. Let us first understand the basic idea of this algorithm in detail.

2.1 Description of the Algorithm

The Ricart-Agrawala algorithm aiming to achieve mutual exclusion in a network is one of the important and famous algorithms in distributed computing domain. Ricart and Agrawalas algorithm generated mutex in computer setting where the nodes communicate through the messages and there is no shared memory. 2*(N-1) messages are sent by the algorithm and the N here refers to the number of nodes within the network. The assumption of this algorithm is that there is an errorless underlying communications network in which transit times may differentiate and the messages may not be delivered in the order sent. Nodes are supposed to run correctly.

For attempting to call for mutex, the node transmits a REQUEST message to all of the other nodes. Once that message is received, then the other node either transmits a REPLY instantly or delays its response until it leaves its own critical section. A node enters its critical section after all other nodes have been notified of the request and have sent a reply granting their permission.

The algorithm relies on the fact that a node that receives the REQUEST message is able to immediately determine whether the requesting node or itself should be allowed to enter its critical section first. The node transmitting the REQUEST massage is never informed about the result of that comparison. Then the REPLY message is sent back immediately if the transmitter of the REQUEST message has the priority; otherwise the REPLY will be postponed.

Through comparing the sequence number in each of REQUEST messages, the order decision is made. In case of the equal sequence numbers, the node numbers will be compared in order to determine the one having the priority.

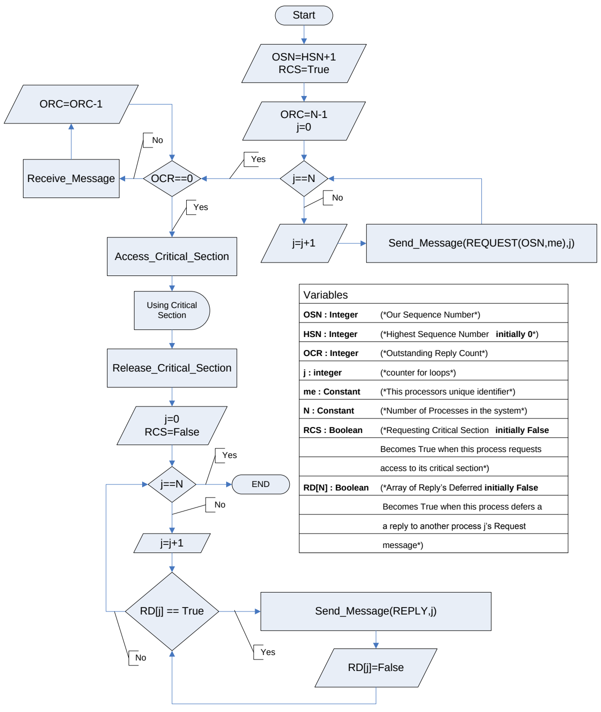
There are N nodes within the network. Each of the nodes performs same algorithm but refers to is own unique node as ME.

The node has three processes to implement the mutual exclusion:

- One is awakened when mutual exclusion is invoked on behalf of this node.
- Another receives and processes REQUEST messages.
- The last receives and processes REPLY messages.

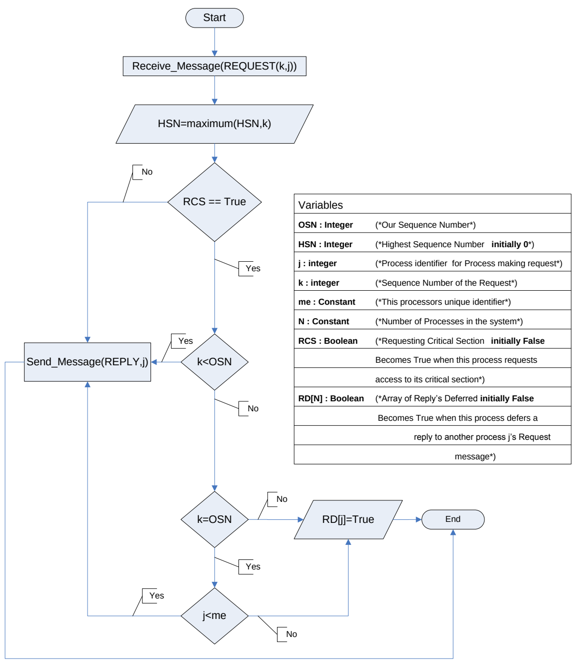
These processes don't operate synchronously, but their operation is based on some shared variables. In order to serialize the access to the shared variables, the semaphore is utilized when needed. If a node is able to create multiple internal mutex requests, then it is expected to own a method for serializing the requests.

From the algorithm, it can easily be understood that there is no possibility of occurrence of any deadlocks and starvations with Ricart and Agrawala's mutex model. The algorithm is depicted in Figure 1 and 2 by flowcharts.



Variables	
OSN : Integer	(*Our Sequence Number*)
HSN : Integer	(*Highest Sequence Number **initially 0***)
OCR : Integer	(*Outstanding Reply Count*)
j : integer	(*counter for loops*)
me : Constant	(*This processors unique identifier*)
N : Constant	(*Number of Processes in the system*)
RCS : Boolean	(*Requesting Critical Section **initially False**
	Becomes True when this process requests
	access to its critical section*)
RD[N] : Boolean	(*Array of Reply's Deferred **initially False**
	Becomes True when this process defers a
	a reply to another process j's Request
	message*)

Figure 1. Flowchart for a process which invokes mutual exclusion in Ricart and Agrawala's mutual exclusion algorithm.



Variables	
OSN : Integer	(*Our Sequence Number*)
HSN : Integer	(*Highest Sequence Number **initially 0***)
j : integer	(*Process identifier for Process making request*)
k : integer	(*Sequence Number of the Request*)
me : Constant	(*This processors unique identifier*)
N : Constant	(*Number of Processes in the system*)
RCS : Boolean	(*Requesting Critical Section **initially False**
	Becomes True when this process requests
	access to its critical section*)
RD[N] : Boolean	(*Array of Reply's Deferred **initially False**
	Becomes True when this process defers a
	reply to another process j's Request
	message*)

Figure 2. Flowchart for a process which receives a request message in Ricart and Agrawala's mutual exclusion algorithm.

2.2 Petri Net Model for Ricart and Agrawala's Algorithm of Mutual Exclusion

A Petri net can be defined as a graphical and mathematical modelling tool. Its parts can be listed as the places, the transitions, and the arcs connecting them. Input arcs link the places with transitions, while output arcs originate from a transition and end at a place. The places may involve tokens; the actual status of the modelled system is determined according to the number of tokens in each place. Transitions are the active components. These components are responsible for modelling the activities that can emerge (the transition fires) and, for this reason, they can affect the state of the system (the important feature of the Petri net). When enabled, the transitions can only fire, and it means that all of the prerequisites of the activity must be satisfied. Once the transition fires, it will take tokens from its input places to its output places. The number of tokens removed / added is determined based on the cardinality of each of the arcs.

Petri nets are accepted as the promising tool for defining and evaluating systems that have special features such as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. Given that they are graphical tools, it can easily be seen that Petri nets can be utilized as a visual-communication solution like flow charts, block diagrams, and networks. Besides that, the reason of the utilization of the tokens in these nets is to simulate the dynamic and concurrent activities of the real systems. As a mathematical tool, it is possible to establish the state equations, algebraic equations, and other mathematical models controlling the behavior of the systems.

While investigating the performance and dependability issues of the systems, it is significantly important to involve the timing concept in the model. Although there are various ways of doing this for a Petri net, the most widely used method is to associate a firing delay with each transition. This delay determines the time that the transition must be enabled, before it can actually fire. When the delay is considered to be a random distribution function, then the net class would be named stochastic Petri net. They types of the transitions can be named based on their associated delay such as the immediate transitions having no delay, the exponential transitions having delay being an exponential distribution, and the deterministic transitions where the delay is fixed.

2.3 Model of a Process in Ricart and Agrawala's Mutual Exclusion Algorithm

The Petri net model for the first module is described in Figure 3. This module takes a message through the network from other processes and after processing these messages it sends a message to the respective processes through the network.

The process receives a message in place S1 and this causes transition X1 to fire. Transition X1 sorts the message and finds out if it is a message that was sent by this process, in which case it is absorbed by placing the token in place S13 and hence firing of transition T13. If this is not a message from this process then it has to be either a REQUEST message or REPLY message. The REQUEST messages are routed to place S11 and the REPLY messages to S12. The Transition T12 is fired whenever a token is placed in place S12 and this transition causes the counter of replies received for this processes request by one. The presence of a token in place S11 implies that a REQUEST for critical section has been made by some process in the system and this causes transition X11 to fire. This transition determines the priority of the REQUEST it this process itself has also made a REQUEST to other processes for access to its critical section. If this process has not made such a request then the REPLY message is sent immediately by transferring the token to place S700 and Firing of Transition Y1000 which passes the token to Place S1000, from where the token is handled by the module for the network (described later). If there is a REQUEST made by this process then the priority of the received REQUEST message is compared by the priority of this processes REQUEST message. The process with smaller sequence number gets higher priority and incase both the processes have the same sequence number then the process with lower process id gets the higher priority. In case the message received has higher priority then the token is again placed in Place S700 and transition Y1000 fires placing the token in place S1000 from where the network module processes it. If the REQUEST message of this processes has higher priority then the transition X11 directs the token to be placed in queue place Q1 meaning that the reply to this message is postponed until this process gets access to critical section and after utilizing its critical section it replies to these so-called deferred messages.

The simulation model assumes that the process in the beginning is busy in doing its main work. This is illustrated in figure 3 by a marking in place S5. The transition T1 is the first transition to fire in this simulation model and depicts that this process is doing the main work. The duration of the main work is simulated by exponential distribution with varying mean time. This mean time is different when the simulation is executed for different number of processes in the system and for different work loads on the processes. After the delay for main work is over the processes always asks for access to critical section and hence a token is placed in place S900 which simulates the creation of REQUEST message by this process and this REQUEST message is propagated through the network by the network module after firing of transition Y1000 and placing of a token in place S1000.

Transition T1 also places a token in place S20 which invokes the transition Y20. Transition Y20 waits for the reply messages from all the other processes to come before allowing this process to access its shared resource. This transition checks if all N-1 replies have been accounted for and then places a token in place S21. Hence the time the token stays in place S20 shows the average waiting time for this process. The presence of a token in place S21 causes the transition T2 to invoke its procedure. This transition is used to show the usage of shared resource by this process. Using of shared resource is represented in this transition by using delay of exponential distribution. Once this exponentially distributed delay is over the transition T2 fires and places the token in place S22. Hence the time the token stayed in place S21 shows the average time this process used the shared resource. Furthermore it can be noted that the combined token time in places S20 and S21 mark the average response time for this process. The transition S22 summons transition Y22 to start.

Transition Y22 collects inputs from place S23 and place S22 and replaces a token in place S24. This causes transition X24 to fire. The transition X24 puts the token in place S5 if there is no deferred reply for other processes (i.e. if Q1 is empty). If Q1 is not empty then it places token in place S25 which causes transition T25 to place token in S800. The presence of a token in place S800 means that transition Y1000 has to forward a reply to processes whose reply was deferred earlier because this process had higher priority for critical section. Transition Y1000 does so by placing a token in place S1000. Once all the deferred messages are replied, token is again placed in place S5 through transition Y22 and transition X24. Figure 3 is shown below for better understanding of this module.

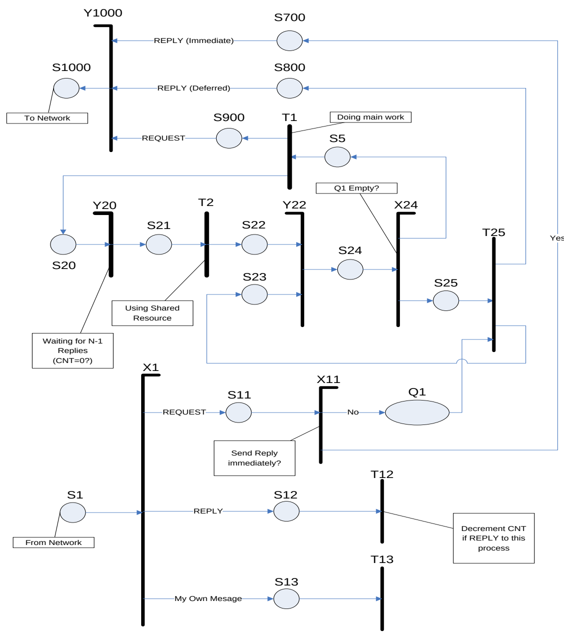


Figure 3. E-net scheme of a process for the Ricart and Agrawala mutual exclusion algorithm.

2.4 Model of a Network in Ricart and Agrawala's Mutual Exclusion Algorithm

This module deals with the handling of the messages in the network and is illustrated by Figure 4. The place S1 is dedicated for the messages generated by process 1, S2 is dedicated for the messages coming from process 2 and hence Sn is reserved for messages coming from process number n. The working of the entire model is based on one routine ad that is when ever a message is generated or posted by any process then it is placed by this module in a fixed S place for that message. Let's assume, for the purpose of explaining this working that process 3 placed a token in its place S1000, then this network module will take that token and place it in place S3, this will cause the firing of transition T3 and hence the token will be taken from place S3 and put into the queue place Q3. Place Q3 contains token, each representing a message from process 3 and is processed by transition Y1 in a cyclic manner. Transition Y1 gives each queue attached to its input location equal chance and places its in all the output places attached to transition Y1, hence it propagates the message to all the processes in the system. Transition Y1 is also used in this simulation model to depict network delay of uniform distribution with mean 2 and variance 3.

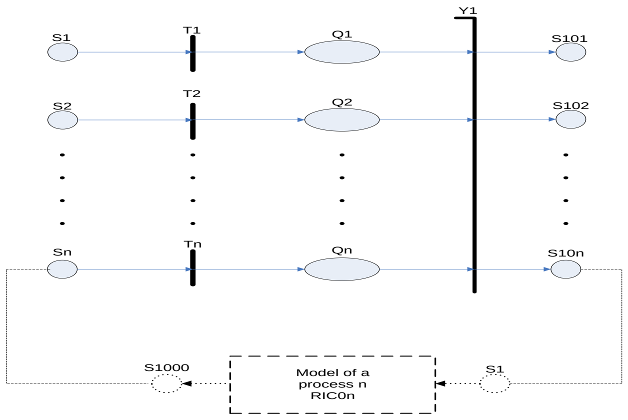


Figure 4. E-net scheme of a network for the Ricart and Agrawala mutual exclusion algorithm.

3. Results and Discussion

In this simulation the most important parameters of simulation for the purpose of understanding the behavior of the algorithm are the average response time for a process to get access to the shared resource and the number of messages passed for a process to get access to the shared resource. It is not desirable for any of these parameters to be high, because if the response time increases for the processes with the increase in the number of processes in the system then it means that the algorithm is less scalable. Furthermore if the number of messages passed by each process to make an access to the shared resource increases with the increase in the number of processes then the system becomes overloaded with messages as the number of processes increase, another undesirable situation.

As it is clear from the simulation results, the model was simulated for varying number of processes in the system under varying loads. The system was tested with 3, 6, 9, 12 and 15 processes at a time with first low then medium and finally high loads.

Table 1 details the Average Response time for all the processes with the different loads and it is obvious from the graph shown in Figure 5 (graph is obtained from table 1) that with low and medium load the system is very scalable as the response time does not change drastically for low and medium load even with 15 processes in the system. However for high load the response time grows almost exponentially (in the worst case) or linearly (at the best), the response time for a process on average with three processes in the system was only 1054ms but with 15 processes in the system with high load it becomes 2148(ms).

Table 1. Average Response time for all the processes with low medium and high loads.

Table 1	**Average Response Time**				
Number of Processes / Load	**3**	**6**	**9**	**12**	**15**
0.1	535.08047 33	555.9 543	573.3 28	574.7 297	587.8 243
0.5	696.4963	813.8 13	875.1 746	914.5 269	938.4 804
0.9	1054.0936	1475. 622	1770. 787	1996. 961	2148. 411

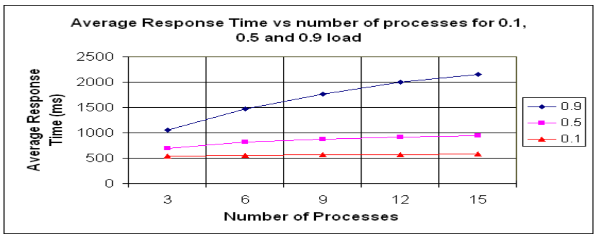


Figure 5. Graph of Average Response Time vs Number of processes for high, medium and low loads.

Table 2 shows the data related to average number of messages per use of shared resource for 3, 6, 9, 12,and 15 processes with low, medium and high load. Figure 6, 7 and Figure 8 illustrate the respective graphs for low, medium and high load performance of the system in relation to average number of messages per use of shared resource. These figures show that the model is highly scalable with respect to average number of messages per use of shared resource, as N for all, 3, 6, 9, 12, and 15 processes.

Table 2. Average number of messages per use of shared resource for all the processes with low medium and high loads.

Table 2	**Average Number of Messages per use of Shared Resource**				
Number of Procersses / Load	**3**	**6**	**9**	**12**	**15**
0.1	3	6	9	12	15
0.5	3.00012 6127	6	9.000 419	12	15
0.9	3.00012 5337	6.000 156	9.000 162	12.00 017	15.00 062

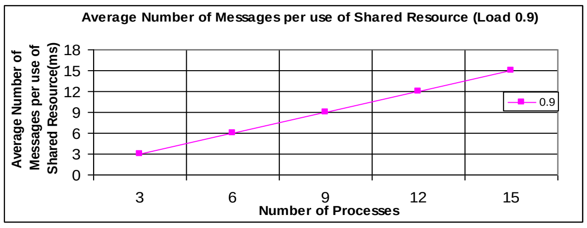


Figure 6. Graph of Average Number of messages per use of shared resource vs number of processes for high load.

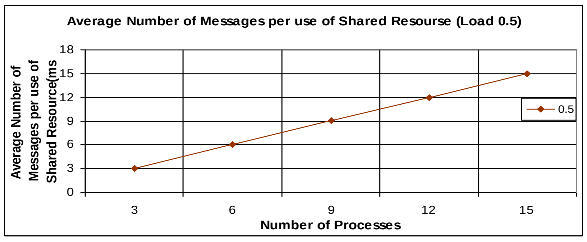


Figure 7. Graph of Average Number of messages per use of shared resource vs number of processes for medium load.

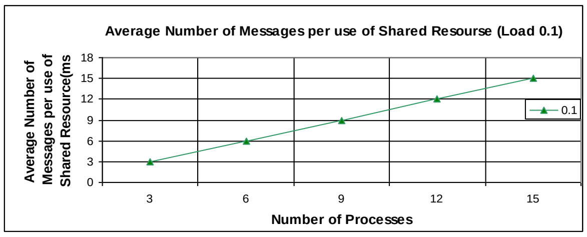


Figure 8. Graph of Average Number of messages per use of shared resource vs number of processes for low load.

4. Conclusion

In this paper, a Petri net based simulation model for Ricart and Agrawala mutual exclusion algorithm was presented. Performance of the algorithm was measured in terms of average response time for a process to get access to the shared resource and the number of messages passed for a process to get access to the shared resource. The model was simulated for varying number of processes in the system under varying loads.

References

Baier, C.; Daum, M.; Engel, B.; Härtig, H.; Klein, J.; Klüppelholz, S.; ... & Völp, M. (2015). Locks: Picking key methods for a scalable quantitative analysis, *Journal of Computer and System Sciences*, Vol. 81, No. 1, 258-287.

Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control, *Communications of the ACM*, Vol. 8, No. 9, 569.

Khanna, A.; Singh, A. K.; Swaroop, A. (2014). A leader-based k-local mutual exclusion algorithm using token for MANETs. *Journal of Information Science And Engineering*, Vol. *30*, 1303-1319.

Knuth, D. E. (1966). Addition comments on a problem in concurrent programming control, *Communications of the ACM*, Vol. 9, No. 5, 321-322.

Kostin, A.; Ilushechkina, L. (2005). Winsim: a tool for performance evaluation of parallel and distributed systems, *Proceedings of the International Conference on Advances in Information Systems,* 312-321.

Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem, *Communications of the ACM*, Vol. 17, No. 8, 453-455.

Lamport, L. (1977). Concurrent reading and writing, *Communications of the ACM*, Vol. 20, No. 11, 806-811.

Lamport, L. (1987). A fast mutual exclusion algorithm, *ACM Transactions on Computer Systems*, Vol. 5, No. 1, 1-11.

Mellor-Crummey, J. M.; Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Transactions on Computer Systems*, Vol. 9, No. 1, 21-65.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, Vol. 77, No. 4, 541-580.

Peterson, G. L. (1981). Myths about the mutual exclusion problem, *Information Processing Letters*, Vol. 12, No. 3, 115-116.

Ricart, G., & Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, *24*(1), 9-17.

Wang, J.; Wang, Z. (2014). Mutual Exclusion Algorithms in the Shared Queue Model, *Proceedings of the International Conference on Distributed Computing and Networking*, 29-43.

Wong, M., Ayguadé, E., Gottschlich, J., Luchangco, V., de Supinski, B. R., & Bihari, B. (2014). Towards Transactional Memory for OpenMP, *Proceedings of the 10th International Workshop on OpenMP,* 130-145.