# Study of Ricart Agrawala Algorithm with modified Ricart Agrawala Algorithm

Rahul Khadse
Computer Science Department
Kent State University
Kent , Ohio
email: rkhadse@kent.edu

***Abstract*—Ricart Agrawala's mutual exclusion algorithm[2] is implemented to conduct experiments to study performance of Ricart Agrawala and Modified Ricart Agrawala. It was expected that original Ricart Agrawala would perform at 2*(n-1) when it comes to message complexity. In modified Ricart Agrawala Algorithm the system increments the Sequence Number by Higher Number (instead of 1) for lower priority processes. Modified Ricart Agrawala was expected to be work same as original ricart agrawala except the fact that higher numbered processes get higher priority for critical section access. Analysis of original and modified algorithm is done using simulation under different number of processes(N) and different contention load sizes (L). Original Ricart Agrawala performs as expected but Modified Ricart Agrawala works exactly same as Ricart Agrawala. The simulation allows analysis of the data in message exchanges required to enter critical section per node and behaviour of algorithm under different conditions.**

## I. Introduction

Implementation of Distributed mutual exclusion(DMX) are of two types : Non Token based and Token based. In token based DMX algorithm mutual exclusion is by using tokens while non token based DMX algorithm uses locks. In Lamport's[1] DMX Algorithm process requesting mutual exclusion sends messages to all processes and waits for reply if it is allowed to enter critical section. Once critical section access is done , it will notify all processes by releasing the request. The Lamport Algorithm uses messages : REQUEST, REPLY, AND RELEASE per critical section. Since three messages are required per critical section, the number of messages required is 3*(*N*-1).

Ricart Agrawala algorithm[2] is optimization of Lamport's Algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages.Since a request is only allowed to enter critical section when process when a process recieves all REPLY messages from all processes, a REPLY can be delayed to a node only when a node is done with it's critical section. Hence number of messages required to enter critical section are reduced from 3*(*N*-1) to 2*(*N*-1).

This report is presented as follows : In section II, we presented the experiment parameters used , the assumptions for realistic simulation engine. In section III we present our findings of running the simulation for various parameters and our analysis of the results .

Finally in section IV , we summarized our findings and presented our future research interest in this area.

## II. Experimentation setup

The objective of this experiment was to implement Ricart Agrawala algorithm and Modified Ricart Agrawala for purpose of understanding the behaviour of the algorithm under different number of nodes(*N*) and different load (*L*). The program implemented was run several times, with different experiment parameters to collect statistical data. This data is used to plot results and analyse the behavior of the algorithms.

### *A. Experiment Parameters and Expectation*

#### *1) Original Ricart Agrawala :*

The parameters used in Ricart Agrawala simulation started by varying the size of the system , number of nodes *(N*) and varying load size (*L*). The number of nodes used varied from 10 to 100. In each increment of 10 nodes , I then varied the size of the contending nodes, load size (*L*) with low load 1(node) and high load (all nodes). It is expected, that in Ricart Agrawala, the change in Load Size will not affect the number of messages being exchanged between nodes. In fact , I expect it to have a constant number of messages exchanged despite change in load. However change in number of nodes should reflect linear change in number of messages per critical section access.

#### *2) Modified Ricart Agrawala*

The parameters used in Modified Ricart Agrawala simulation started by varying the size of the system , number of nodes (*N*) , varying sequence number increment for lower priority processes. In each case I varied the size of system by varying number of nodes(*N*), and sequence number increment , and observed number of Critical Section accesses made by High

priority and low priority processes when total number of Critical section accesses are same. Each process given a fair chance to execute the critical section . I expected to have same number of critical section accesses for both high priority and low priority processes irrespective of change in sequence number increment. Because when any process sends message with sequence number to any process , it updates its sequence number if received sequence number is greater than its sequence number. So even if high priority process's sequence number is greater than that of low priority process's sequence number, the high priority processes will eventually update its highest sequence number after receival of request . Ultimately total number of critical section accesses are going to be same. Only order of exceution by processes will change.

### *B. Simulation Engine*

#### *1) Original Ricart Agrawala*

The simulator implemented is responsible to handle creation of nodes , selection of nodes by random to be run , and assigning channels to the nodes. I used random number generator to generate a random process to be run . In this way simulations were random such that no two runs have the same computations.

To ensure that the simulation executed as according to the specifications of the algorithm, receive request function should be implemented such that when process i recives a request from process j , it sends a REPLY message to process i, if process j is neither requesting nor executing the CS or if process j is requesting and then process j's own request timestamp. This implementation is done using function rec_request as following figure(1):

```
void process::rec_request(int k,int j ,vector<process>* J,int N)
{//receive request using this function
mu();
bool deferit;
deferit=false;
cout<<"Process "<<id<<" received request from process"<<j<<"with SQN :"<<k<<endl;
cout<<"Process P["<<id<<"] receving request of Process P["<<j<<"]"<<endl;
cout<<"P["<<id<<"] 's "<<"highSQN:  "<<highSQN<<"& received SQN ="<<k<<endl;
if(j>id)
cout<<"and "<<id<<"<"<<j<<endl;
highSQN= max(highSQN,k);//update highest sequence no !
//cout<<"highSQN:  "<<highSQN<<endl;
deferit=((requestingCS)&&((k>mySQN)||((k==mySQN)&&(j>id))));
if(deferit == true)
{
    replydeferred[j]=true;
    cout<<"Process P["<<id<<"] deferred reply for Process P["<<j<<"]"<<endl;
}
else
{
cout<<"Process P["<<id<<"] sending reply to Process P["<<j<<"]"<<endl;
reply(j,J,N);
}
mu();
}
```

Figure (1)

#### *2) Modified Ricart Agrawala*

Original Ricart Agrawala Algorithm tends to favor lower numbered nodes slightly using to tie breaking rule. This favouritism can be reduced by incrementing the sequence number by higher number for modified Ricart Agrawala . To implement this I used the algorithm given in figure 2. The sequence number for lower numbered process should be incremented by higher number and vice versa .

```
if(id<(N/2))//lower numbered process or higher priority
mySQN=highSQN + increment;
else
mySQN=highSQN + 1;//for lower priority processes
```

Figure (2)

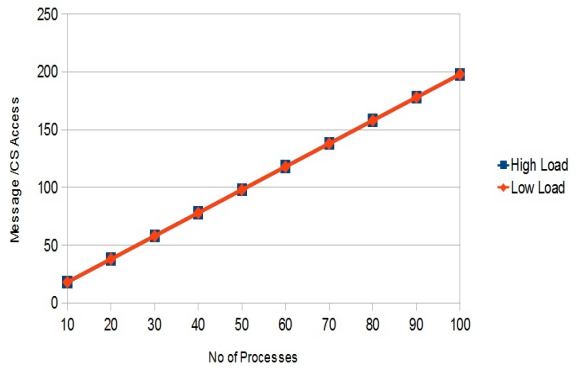
## III. RESULTS

### *A. Original Ricart Agrawala*

The results of the experiment for Ricart Agrawala Algorithm for varying number of nodes N and varying load size is shown in figure (3). It can be seen from the table that for any given fixed number of nodes N , varying the Load size doesnot change the number of messages being exchanged in the system.

| *No of Processes* | *High Load* | *Low Load* |
|---|---|---|
| *10* | 18 | 18 |
| *20* | 38 | 38 |
| *30* | 58 | 58 |
| *40* | 78 | 78 |
| *50* | 98 | 98 |
| *60* | 118 | 118 |
| *70* | 138 | 138 |
| *80* | 158 | 158 |
| *90* | 178 | 178 |
| *100* | 198 | 198 |

Figure(3)




Figure(4)

Figure(4) shows a graph of number of processes versus number of messages per CS accesses under high load and low load . The graph shows that message complexity doesnot change in case of high load and low load. It remains same. Message Complexity changes linearly with number of processes in the system. This linear change is exactly 2*(*N*-1) where *N* is number of nodes. This results are as expected because each node requires to send exactly *N* number of nodes twice for request and reply . Hence the graph shows a linear growth with increase in number of processes regardless of load size.

### *B. Modified Ricart Agrawala*

The algorithm tends to favor lower numbered nodes slightly owing to the tie breaking rule[2]. This favoritism can be reduced incrementing the sequence number of low priority process by larger integer.

Fig(5) shows table which contains total number of processes. Increments used for higher numbered processes, number of CS accesses made by high priority(lower numbered processes) and lower priority(higher numbered processes when total number of CS accesses are 200. I calculated difference between number of CS accesses made by high priority and lower priority processes. This difference happens to be zero. Fig (6) shows analysis done using same concepts on ten number of processes. Irrespective of number of increments ;number of CS accesses made by each process was same when total number of CS accesses was 200.

| No of Processes | Increment for Higher numbered nodes | no of CS acesses made by High Priority (Lower numbered processes) | no of CS acesses made by Low Priority (Higher Numbered processes) | Difference |
|---|---|---|---|---|
| 2 | 1 | 100 | 100 | 0 |
| 10 | 1 | 100 | 100 | 0 |
| 10 | 5 | 100 | 100 | 0 |
| 20 | 10 | 100 | 100 | 0 |
| 20 | 20 | 100 | 100 | 0 |
| 50 | 20 | 100 | 100 | 0 |
| 100 | 20 | 100 | 100 | 0 |
| 200 | 20 | 100 | 100 | 0 |

Fig(5)

| | *High Priority Process* | | | | | *Low Priority Process* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *Increment* | *P0* | *P1* | *P2* | *P3* | *P4* | *P5* | *P6* | *P7* | *P8* | *P9* |
| *1* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *2* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *3* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *5* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *10* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *50* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *100* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *200* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *500* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| *1000* | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |

Fig(6)

## IV. Conclusion

From the data analysed the results we can conclude that message complexity of Ricart Agrawala is not depend on high load or low load but depends on the number of processes in the system. Thus Ricart Agrawala gives constant performance even in case of high load.

In modified Ricart Agrawala the message complexity of the system will not be affected. Irrespective of sequence number increment by higher number for low numbered (Higher Priority Process) , the number of CS accesses made by high priority and low priority processes are same.

Ricart Agrawala can be extended to work on practical network applications. In practical network insertion of new nodes to the network working on mutual exclusion should not affect the correctness of algorithm. The newly added nodes should be able to update their sequence number , request received , and should be able to determine replies to deployed.

Ricart Agrawala can be extended to solve Dining Philosopher's Problem where there are several sites and several number of processes working in each site.

## References


[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978.

[2] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," Commun. ACM, vol. 24, no. 1, pp.9–17, 1981.

[3] http://deneb.cs.kent.edu/~mikhail/classes/aos/