# Last Entered Algorithm for Mutual Exclusion in a Distributed System

Hridesh Rajan
Lucent Technologies
hridesh@lucent.com

## Abstract

*Mutual exclusion in the distributed systems, components of which are connected by a ring topology (Logical or Physical) network is sometimes realized by employing a token to determine the process among those which compete for the critical section. Usually the token circulates freely around the network as a result accesses to such a mutual exclusion lock may create heavy synchronization traffic and/or serious contention over the network, thereby degrading system performance considerably. In this paper, a scheme that keeps the synchronization traffic low is introduced. The scheme is based upon master-slave model in which the master is dynamic. This is achieved by using a variable in each of the processes to know which process last entered the critical region. Proposed algorithm was simulated as well as implemented and the results indeed reduced the synchronization traffic and increased performance.*

## 1. Introduction

The small part of a program, which competes for access to the shared resources and thus requires arbitration, is called critical section. Mutual-exclusion is guaranteeing that critical sections will not be executed by more than one process simultaneously. In the case of a single processor solution to the problem of mutual exclusion is usually achieved by guaranteeing that no context switch takes place during the critical section. In case of distributed systems where some kind of concurrency may exist among the processes this solution fails.

The most obvious solution in the case of distributed systems is to establish a logical ring of processes in the system and circulate a token over it, which will arbitrate the access to the critical region. The token circulates around the ring and passes through a node even if it does not wish to enter the critical region. In the mean time the process wishing to enter the critical region has to wait for the token to arrive after circulating through few idle nodes (from this point onwards this time will be referred as waste-time). In case of larger systems this waste-time can be significant. In addition free circulation of token around the network may create heavy synchronization traffic and serious contention over the network.

For mutual exclusion algorithms to operate with a minimum of waste-time and network traffic in a fairly large system a solution could be to form a dynamic queue of processes within the big logical ring which will consist of the processes wishing to enter the critical region. Rest of the processes is spared of participating in the unnecessary communication thus resulting in a decrease in synchronization traffic as well as the waste time. This algorithm achieves this by using a data-structure “LAST-ENTERED” in each of the processes to know which process last entered the critical region. It also achieves access on First-Come-First-Serve basis. The philosophy of this algorithm is KISS (Keep It Simple, Stupid).

The remainder of this paper is organized into four sections. Section 2 reviews the various mutual exclusion algorithms in place. Section 3 describes the algorithm Section 4 evaluates the algorithm and section 5 concludes.

## 2. History of Mutual Exclusion Algorithms

T. Dekker (1962): Proposed problem of multiprocessor mutual-exclusion.

Edsger Dijkstra (1965): First Solution [2] guaranteed mutual exclusion, freedom from deadlock, and freedom from livelock, but

allowed the possibility of one process being forever stuck in the entry section while other processes are allowed into the critical section.

Donald Knuth (1966): Guaranteed freedom from starvation [4].

Courtois, Heymans, and Parnas (1971): Proposed two algorithms [1] that allowed concurrent reads while ensuring that writes are exclusive from each other and from reads.

Leslie Lamport (1974): Published a solution [5] in which processes were served in a first-come first-served order.

Leslie Lamport (1977): Suggested an algorithm [7], which would always allow a single writer to write without blocking.

James Burns (1978): Proposed an algorithm, [15] that uses binary-shared variables, it solves the problem but does not guarantee any fairness.

Thomas (1979): Proposed an algorithm which [13] requires a majority of the processors to consent.

Fischer, Lynch, Burns and Borodin (1979): proposed an algorithm [16], which uses nxn shared variables.

Gary Peterson (1981): Published algorithm [11] for two processes, and also gave an algorithm for more than two processes.

Ricart and Agrawala's (1981): Proposed an algorithm [12], which used messages. This algorithm required N-1 messages to ask for permission and N-1 to grant it.

Leslie Lamport (1987): ``fast" mutual exclusion algorithm [8], which only required seven memory accesses in the event of no contention.

Mellor-Crummey and Scott (1991): Proposed an algorithm [10] that had remote references.

Yang and Anderson (1993): Proposed a algorithm [14] which only required atomic reads and writes.

Maekawa (1985): Proposed an algorithm [9] in which groups of processors were created and each processor were in a group.

## 3. Algorithm

This algorithm is divided into three parts:
1. Function to enter the critical region (ENTER).
2. Function to leave the critical region (LEAVE).
3. Function executed in the background to keep the ball rolling (DAEMON).

The language used in statement of the algorithm is pseudo ‘c’.

### 3.1. Data Structures:

Message structure:

```
struct   MESSAGE {
int TYPE;
int SENDER;
int QUEUE[];
int QUEUE_LENGTH;
                    };
```

### 3.2. Messages Types

1. WISHING_TO_ENTER:
Message description: Used when a process wishes to enter a critical region.
Member variables: Process ID of the Sender.

2. ALREADY_THERE:
Message description: Used when a process is in a critical region and it receives a WISHING_TO_ENTER message.
Member variables: Process ID of the Sender.

3. REGION_TRANSFER:
Message description: Used by a process done with the critical region to transfer the access token to the next process in the queue.
Member variables: Process ID of the Sender, wait queue for the critical region.

### 3.3. Stub Functions used in the algorithm

Function: Broadcast
Arguments: Message type, Data.
Return value: none.
Description: This function sends this message to every alive process.

Function: Send
Arguments: Receiver’s ID, Message type, Data
Return value: Status.
Description: This function sends this message to the Receiver and returns the outcome as SUCCESS or FAILURE.

Function: Wait
Arguments: No of quanta of time to wait and /or type of message expected
Return value: Message Received
Description: this function returns either on time-out or on receipt of some message. If this function is called with argument INFINITY it effectively blocks for a message. If this function is called with a time and a Message type it returns when either there is a timeout or a message of the type specified is received.

Function: Error
Arguments: None
Return value: None
Description: this function is a error handler it insures that no-one is in critical state and the system is rolled back to the last log.

Function: Start_daemon
Arguments: None
Return value: Result of the process
Description: Starts the DAEMON process and returns.

### 3.4. Variables and data structures

LAST_ENTERED: Variable, which holds the process ID of the process, which last entered the critical region. When the process starts this variable is initialized to NULL to assure that it does not contains any garbage value.

PID: process ID of the process calling the function.

Mesg: variable of the type struct MESSAGE.

QUEUE: Array which each member of which is of type process ID and stores the processes in the queue for critical region.

QUEUE_LENGTH: No of elements in the array QUEUE.

REPEAT: integer variable, which represents the no of quanta, the algorithm waited.

TIME_LIMIT: no. of quanta's a process must wait before entering the critical region.

### 3.5. Function ENTER

```
while (1)
{
/* Keep trying to enter the critical region */
if( LAST_ENTERED == NULL )
/* If the value of this variable is NULL i.e. the process has just started so it must poll the existing group to know the status */
        {
Broadcast (WISHING_TO_ENTER, PID);
/* The process then broadcasts a WISHING_TO_ENTER message indicating its wish to enter the critical region */
        Mesg = Wait(1);
/*It then waits for either one time quantum or any message from the rest of the group */
        switch ( Mesg . TYPE)
/*Depending on the event it takes action. Type of the event is determined by TYPE variable of the data structure message, which is returned by the wait function */
                {
                case ALREADY_THERE:
/* Received a ALREADY_THERE message i.e. there is some process which is already in the critical region
                LAST_ENTERED=
                Mesg.SENDER;
/*Record that the sender process is in the critical region now.*/
                break;
                case WISHING_TO_ENTER:
/*Received a wishing to enter message, i.e. any other process is also interested in entering the critical region */
                if( Mesg.SENDER > PID )
/* if process id of the sender is greater than that of the receiver */
                LAST_ENTERED=
                Mesg.SENDER;
/*Relinquish control and let the leader enter the critical region */
/* other wise if the sender's process id is less than the receiver's process-id, receiver is entitled for the critical region. */
                break;
                case REGION_TRANSFER:
/* some process is trying to transfer the access of the critical region to the receiver by mistake so discard the message*/
                break;
                case TIME_OUT:
/* The wait timed out after one quanta of time */
                if(REPEAT >= TIME_LIMIT)
                return;
```

```
/* Waited sufficiently long so let us enter the
critical region */
                REPEAT=REPEAT+1;
/* waited one more quanta */
                break;
                }/* End of switch loop */
        }/* End of if scope */
else
/* If the value of LAST_ENTERED is not
NULL */
        {
        if((send(LAST_ENTERED,
        WISHING_TO_ENTER,PID))==
        FAILURE)
                {
                LAST_ENTERED = NULL;
                Break;
                }
/* The LAST_ENTERED variable now contains
the Process ID of the process, which is currently
in the critical region. Send a
WISHING_TO_ENTER message to the process.
If the send is successful go ahead otherwise
search for the process in the critical region. */
        Mesg = Wait(INFINITY);
/*Wait for any message */

switch ( Mesg . TYPE)
/*Depending on the event it takes action. Type of
the event is determined by TYPE variable of the
data structure message, which is returned by the
wait function */
                {
                case ALREADY_THERE:
/* Received a ALREADY_THERE message i.e.
there is some process which is already in the
critical region
                LAST_ENTERED=
                Mesg.SENDER;
/*Record that the sender process is in the critical
region now.*/
                break;
                case WISHING_TO_ENTER:
/*Received a wishing to enter message, i.e. any
other process is also interested in entering the
critical region. We are already in the queue for
critical region so discard the message. */
                break;
                case REGION_TRANSFER:
/* some process is trying to transfer the  access
of the critical region .Now the message contains
a queue of processes for critical region. */

                if((send(Mesg.SENDER,
                ALREADY_THERE,PID)) ==
                FAILURE)
                        {
                LAST_ENTERED = NULL;
                        Break;
                        }
/* First acknowledge the receipt of the message
to the sender. */
                for(COUNT=1 ;
        COUNT < LENGTH_OF_QUEUE;
                COUNT++)
                send(Mesg.Queue[COUNT],
                ALREADY_THERE,PID));
/*Refresh the LAST_ENTERED variable of each
of the processes in the Queue for critical region.
*/
                LAST_ENTERED = PID;
/*Now set the LAST_ENTERED to itself */
                return;
/* NOW IN CRITICAL REGION */
                break;
                }/* End of switch scope */
        }/* End of else scope */
}/* end of the while loop */

/*End of function ENTER */
```

### 3.6. Function LEAVE

```
Variable: type integer COUNT=1;
while(1)
{
/*Keep trying until in critical region */
If(COUNT < LENGTH_OF_QUEUE)
/* There are some processes waiting in QUEUE
to enter the critical region */
        {
        send(QUEUE[COUNT],
        REGION_TRANSFER,PID);
        /* Try to transfer the critical region to
        the process waiting in the queue at
        position count. */
        COUNT = COUNT+1;
        /* Increment count so that next time try
        to transfer critical region to some other
        process */
        Mesg = Wait(1);
/*Wait for a message or one quantum of time */
        switch ( Mesg . TYPE)
/*Depending on the event it takes action. Type of
the event is determined by TYPE variable of the
data structure message, which is returned by the
wait function */
                {
                case ALREADY_THERE:
/* Received a ALREADY_THERE message i.e.
the critical region transfer  is successful */
LAST_ENTERED= Mesg.SENDER;
```

```
/*Record that the sender process is in the critical region now.*/
                    return ;
                    /* Job finished so return */
                    break;
                    case WISHING_TO_ENTER:
/*Received a wishing to enter message, i.e. any other process is also interested in entering the critical region. We have a queue for critical region so add the sender to the queue. */
                    LENGTH_OF_QUEUE=
                    LENGTH_OF_QUEUE + 1;
                    QUEUE
                    [LENGTH_OF_QUEUE] =
                    Mesg.SENDER;
                    break;
                    case REGION_TRANSFER:
/* some process is trying to transfer the access of the critical region .Now this is failure. */
                    Error();
                    break;
                    }/* End of switch */
          }/*End of if scope */
else{
/*if there are no processes in the queue */
          LAST_ENTERED=PID;
/* store information about the last process entering the critical region */
Start_Daemon ();
/* Start the Daemon function to take care of the critical region */
return;
/*Exit critical region */
}/*end of else scope */
}/* end of while loop */

/*End of the function LEAVE */
```

### 3.7. Function Daemon

```
While(1)
{
/* Till the job is not finished */
Mesg = Wait(INFINITY);
/*Wait for a message */
          switch ( Mesg . TYPE)
/*Depending on the event it takes action. Type of the event is determined by TYPE variable of the data structure message, which is returned by the wait function */
                    {
                    case ALREADY_THERE:
/* Received a ALREADY_THERE message i.e. there is some process in critical region this is a failure */
                    Error();
                    break;
                    case WISHING_TO_ENTER:
/*Received a wishing to enter message, i.e. any other process is interested in entering the critical region. Pass the control of critical region to the sender */
                    send(Mesg.SENDER,
                    REGION_TRANSFER,PID,
                    NULL);
                    /* Try to transfer the critical region to the process */
                    Mesg = Wait(1,
                    ALREADY_ THERE );
                    if(Mesg.TYPE!=
                    TIME_OUT)
                    Return;
                    /* if the returned message is not a time out message i.e. critical region is transferred*/
                    {
                    LAST_ENTERED=
                    Mesg.SENDER;
                    Return;
                    }
                    break;
                    case REGION_TRANSFER:
                    /* some process is trying to transfer the access of the critical region .Now this is failure. */
                    Error();
                    break;
                    }/* End of switch */

}/*end of while loop*/
```

## 4. Complexity Analysis

Let us assume,
N = No of processes in the system.
Q = No of quanta's of time a process waits before entering the critical region.
Maximum number of messages sent per entry
= N*Q + N - 2 (Assuming that the N messages are sent in each broadcast by Process 0 and there is no response till the last wait time and at the last wait quantum a ALREADY_THERE message is received from Process N which has N-2 processes in its crtical region queue.)
Maximum number of messages sent per exit
= N-2 (Assuming that there are N-1 processes in the queue and only last one responds to the REGION_TRANSFER message)
Worst case communication complexity:
O( N*Q) messages required for ENTER and O(N) messages required for LEAVE

Best case communication complexity
O(1) messages required for ENTER and LEAVE

## 5. Conclusion

This paper presented Last Entered Algorithm for mutual exclusion in a Distributed System. The algorithm achieves mutual exclusion through a variable LAST_ENTERED and guarantees access to shared data in a First-Come-First-Serve basis with a communication cost of the order of N*Q (where N is the number of processes and Q is the number of time quantum system is configured to wait for a reply) as opposed to N*N in case of quorum based algorithms.

## 6. References


**1.** Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with ``readers'' and ``writers''. Communications of the ACM 14, 10 (October 1971), 667--668.

**2.** Dijkstra, E. W. Solution of a problem in concurrent programming control.
Communications of the ACM 8, 9 (September 1965), 569.

**3.** Herlihy, Maurice. Wait-free synchronization. ACM Transactions on Programming Languages and Systems 11,1 (January 1991), 124--149.

**4.** Knuth, Donald E. Addition comments on a problem inconcurrent programming control. Communications of theACM 9, 5 (May 1966), 321--322.

**5.** Lamport, Leslie. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17, 8 (August 1974), 453--455.

**6.** Lamport, Leslie. On interprocess communication, Part II: Algorithms. Distributed Computing 1, (1986),86--101.

**7.** Lamport, Leslie. Concurrent reading and writing. Communications of the ACM 20, 11 (November 1977),806--811.

**8.** Lamport, Leslie. A fast mutual exclusion algorithm.ACM Transactions on Computer Systems 5, 1 (February1987), 1--11.

**9.** Maekawa, Mamoru. A algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems 3, 2 (May 1985), 145--159.

**10.** Mellor- Crummey, John M. and Scott, Michael L. Algorithms for scalable synchronization on shared -memory multiprocessors. ACM Transactions on Computer Systems 9, 1 (February 1991), 21--65.

**11.** Peterson, G. L. Myths about the mutual exclusion problem. Information Processing Letters 12, 3 (13 June1981), 115--116.

**12.** Ricart, Glenn and Agrawala, Ashok K. An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM 24, 1 (January 1981), 9--17.

**13.** Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4, 2 (June 1979),180--209.

**14.** Yang, Jae-Heon and Anderson, James H. A fast, scalable mutual exclusion algorithm. Proceedings of the 12thAnnual ACM Symposium on Principles of Distributed Computing, ACM, New York (August 1993), 171--182.

**15.** Burns James E. Mutual exclusion with linear waiting using binary shared variables.ACM SIGACT News 10 (2):42-47,summer 1978.

**16.** Fischer Michael J., Lynch Nancy A., Burns James E., and Borodin Allan : Distributed FIFO allocation of identical resources using small shared space. ACM Transactions on Programming Languages and Systems, 11(1):90-114, January 1989