An Experimental Comparison of Lock-based Distributed Mutual Exclusion Algorithms

Victor E. Lee

Abstract—**The message complexity of two lock-based distributed mutual exclusion algorithms are compared in simulation. The number of messages per critical section invocation is measured in a simulated system as a function of the number of processes *N* and as a function of the critical section contention load *L*. Both algorithms perform as expected, with Ricart and Agrawala's algorithm exhibiting 2*(*N*-1) behavior, and Maekawa's algorithm showing a $K * \sqrt{N}$ dependence, where K is a weak function of load *L*.**

Index Terms—**Distributed Algorithms, Mutual Exclusion, Message Passing Distributed Systems**

I. Introduction

Two basic approaches are used for achieving mutual exclusion in distributed systems: locks and tokens. Lamport presented the first lock-based solution, alongside his introduction of partially ordered events with logical clocks [1]. Lamport's mutual exclusion algorithm applies to fully connected networks with in-order deliver of messages. It has a message complexity of 3*(*N*-1), where *N* is the number of processes.

Other authors have sought to improve upon the message complexity. Ricart and Agrawala presented an algorithm [2] that they argue has a maximum message complexity of 2*(*N*+1). They achieve the reduction by reducing the number of messages exchanged between a requestor and another process. Rather than sending a REPLY immediately after receiving a REQUEST, as in Lamport's algorithm, a process will only do so when it ready to grant its critical section lock to the requestor. This selective use of REPLY eliminates the need for the RELEASE message.

Instead of reducing the number of messages per process pair, Maekawa reduces the number of processes that must be contacted in order to assure mutual exclusion [3]. Each process possesses a set of neighbors, often referred to as its quorum. Quora have the following properties: The quora of any two processes in the system have a non-zero intersection, and a process belongs to its own quorum. A process will only grant its critical section lock to one member of its quorum at a time. Consequently, if a process succeeds in acquiring locks from its entire quorum, it has guaranteed mutual exclusion for the entire system.

This reduces the message complexity to *K***Q*, where *K* is the average number of messages exchanged between a pair of processes, and *Q* is the quorum size. Maekawa argues that the optimal size for Q approaches $\sqrt{N}$, when quora are be configured so that the size of the intersection set between any two processes is one. If all locks are available when a process makes its request, then *K*=3 (REQUEST, GRANT, RELEASE). If a lock is not currently available, then one each of up to three additional messages may be employed (FAIL, INQUIRE, YIELD). Sanders provides details on the proper used of FAIL, INQUIRE, and YIELD [4] under some conditions not discussed by Maekawa. Therefore, *Kmax* = 6. *K* will tend to increase as the load (number of processes simultaneously contending for the critical section) increases.

The remainder of this paper is organized as follows: Section II presents the experimental hypotheses. Section III describes the system model and simulator uses for these experiments. Section IV presents and discusses the results of the experiments. Section V concludes the paper with some suggestions for future work.

II. Hypotheses

Message complexity *M* for lock-based mutual exclusion algorithms is expected to vary with *N* and with load *L*. The goal of this work is to verify, illustrate, and compare the message complexity of the Ricart-Agrawala and Maekawa algorithms. The following hypotheses formally state the expected outcomes:

H1. For a fixed load, *M* in Ricart-Agrawala varies linearly with *N*.
H2. For a fixed load, *M* in Maekawa varies according to $\sqrt{N}$.
H3. For a fixed N, *M* in Ricart-Agrawala does not vary with L.
H4. For a fixed N, *M* in Maekawa is equal to *K***Q*, where *K* gradually increases, from *K*=3 at minimal load to *K*=6 at maximal load.

III. System Modelling and Simulation

The algorithms in this study naturally adhere to the message passing model of distributed systems. Interleaving execution is assumed. Messages may contain sequence numbers, to

Manuscript received December 9, 2007.

Victor E. Lee is a Ph.D. student at the Department of Computer Science, Kent State University, Kent, OH 44242 USA (e-mail: vlee@cs.kent.edu).

implement a logical clock scheme.

A. *Modeling a Distributed System*

A simulator was implemented in Java to compare these two algorithms. The simulator employs generic class structures to represent a Process, a Message, and a Channel. The topology of a network is expressed by providing each Process with a list of its neighboring Processes. The list of neighbors was chosen rather than a global connectivity matrix because it more closely models the algorithms being considered. An abstract Process can Send and Receive messages but has no program to drive these actions.

To implement a Process for a specific algorithm, one defines the Send and Receive functions, message contents, plus any internal actions and state variables. The Channel may be configured to deliver either in-order or out-of-order messages. A complete system is an array of Processes which are assigned their neighbor lists. A Ricart-Agrawala Process has a complete list of all N-1 other processes. A Maekawa Process has a neighbor list containing its unique quorum.

B. *Constructing Quora*

Maekawa bases his quorum on finite projective planes but does not offer an algorithm for computing quorum membership. This study uses the billiard quorum construction algorithm advanced by Agrawal, Eğecioğlu, and Abbadi [5]. In their system, $Q = \sqrt{2N+1}$, so while $\sqrt{N}$ dependence is maintained, quorum size is increased by a constant factor $\sqrt{2}$.

C. *Implementing Message Channels*

For historical reasons, the simulator originally implemented the many process-to-process channels as a single global queue of messages. Messages were delivered according to a global FIFO protocol. This proved to be unacceptable for this study. Ricart-Agrawala supports out-of-order delivery. Maekawa, on the other hand, requires that messages in the local channel between a pair of processes be delivered in order, but messages in different local channels can be delivered concurrently; that is, there is no causal ordering between messages in different channels. To generate randomized computations, the simulator must not insert any ordering that is not required by the algorithms themselves.

Rather than implementing the full $N*Q$ individual channels required by Maekawa, their behavior was emulated with only N queues. The Channel module has N queues, categorizing messages according to sender. Ordering is determined during the Send operation. If delivery may be out-of-order, then Send inserts a message into its sender's queue in a random position. If delivery must be in order, then Send searches that queue for the most recently added message addressed to the given receiver, and then inserts the new message somewhere between this position and the tail of the queue:

```
queue := Queue[sender_id];
index := queue.length;
while (index >= 0 ^
      queue[position].recvr_id != new_msg.recvr_id)
   begin
      index := index – 1;
   end;
{insert new_msg in random position between tail and
      index}
```

Fig. 1. Channel Send function

```
messageCount := csCount := 0;
initializeCSreq(N, L);

while (numCSexits < maxCSexits ^ Channel.notEmpty)
begin
   if (inCS ^ ProbabilityOfExiting) then begin
      csCount++;
      CSExit;   /* tell process to exit CS */
      CSReq;   /* select a random proc. to request CS */
   end;
   if (Channel.notEmpty) then begin
      Message m := Channel.remove;
      messageCount++;
      Deliver(m);
   end;
end;
```

Fig. 2. Simulator Execution Cycle

To emulate deliver of an arbitrary message, Receive selects a random sender queue. If that queue is empty, it tries again. It then removes the message at the front of the queue. Note that randomization of each queue's order was already established during the Send phase.

D. *Roles of the Simulator and Execution Cycle*

The simulator has several roles. The first role below is algorithm-specific. The other roles are generic:

- System Constructor – makes an set of N algorithm-specific processes, then assigns neighbors
- Critical Section Manager - commands a process to request the CS or to exit the CS.
- Load Manager – selects enough random processes to maintain the contention load at the target level
- Accountant – counts messages and CS cycles

When the simulator is invoked, the user specifies parameter values for algorithm choice, number of processes N, load L, and the maximum number of CS cycles C. The simulator first acts as System Constructor. The Load Manager then selects L processes, which the Critical Section Manager instructs to request the CS. All of these initial requests are timestamped with sequence number 0.

Note that a Process acts either when it receives a command from the Critical Section Manager or when it Receives a Message from the Channel. The System Constructor triggers L processes to broadcast REQUEST messages, which will all be queued in the Channel.

The simulator now enters the execution loop. Each cycle, there is a fixed probability (default = 50%) that a process already in the CS will exit the CS. If exit is selected, the CS Manager directs the exit, and the Load Manager selects another process to replace this one. The simulator then selects a random message to deliver. This loop is repeated until the Accountant notes that we have reached the maximum number of CS exits and the Channel queues are empty.

IV. EXPERIMENTAL RESULTS

Two series of simulations were run, with ten trials per test condition per algorithm. One run includes 100 CS cycles. The first series varied *N* while using a minimum load of one process in CS contention. The billiard quorum algorithm only produces automatic results when *Q* is odd. This quantizes values for *N*. Table 1 shows the first four values of *Q* and *N* (excluding *Q* = 1). These values span one order of magnitude for *N* and were the values used for the simulations.

Q	***N***
3	4
5	12
7	24
9	40

Table 1. Billiard Quorum Size and Corresponding Maximum System Size

The second series varied load *L* in a 40-process system. Three loads were used: minimum (one process), half load, and full load (all processes). Again, there were ten runs with 100 CS cycles per run.

Fig. 3 shows the results from the first series: message complexity vs. Number of processes *N*. The message complexity for the Ricart-Agrawala algorithm matched expectations exactly: *M* = 2*(*N*-1) with zero variance.

The message complexity for the Maekawa algorithm exhibits a trend that very closely resembles the $\sqrt{N}$ behavior that was predicted. The variance is extremely small. The actual quorum formula is $Q = \sqrt{2N + 1}$, so for small values of *N*, we expect some divergence from the $\sqrt{N}$ trend. A fairer measure might be to compare message complexity to actual *Q*, to see if under light load conditions, *K* is a constant.

Maekawa (vary *N, L* = 1)				
N	4	12	24	40
Avg Msgs/CS	9.79	16.03	22.02	27.89
Q	3	5	7	9
K	3.26	3.21	3.15	3.10

Table 2. Ratio *K* between Message Complexity and Quorum Size (light load)

The data show that K is not constant but relatively stable and approaches the predicted value of 3 as N increases.

Fig. 4 shows the results from the second series: Message complexity vs. Load. The results for the Ricart-Agrawala algorithm are not shown because they were exactly as predicted and not deemed interesting: for *N*=40, *M*=78 for all load levels. The Maekawa algorithm results are qualitatively similar to what was expected (a small increase), but quantitatively different.

Maekawa (vary *L*, *N* = 40)			
N	1	20	40
Avg Msgs/CS	27.89	35.71	36.00
Q	9	9	9
K	3.10	3.97	4.00

Table 3. Ratio *K* between Message Complexity and Load Level (*N* = 40)

The prediction was for K to range from a minimum of 3 to a maximum of 6. The measured range was from 3.10 to 4.00.

The difference between the predicted results and the measured results are easily explained if the simulation logs are studied. Even when only one process is in contention for the CS, some FAIL, INQUIRE, and YIELD messages may be sent, increasing K above the baseline of 3. Suppose a process exits the CS and broadcasts RELEASE. Due to the interleaving execution semantic and concurrency between different processes, some processes will certainly receive the RELEASE before others. If one of the early recipients is the next contender for the CS and broadcasts its REQUEST, a third process might receive this new REQUEST before receiving the first process' RELEASE. This third process will reply to the second REQUEST with FAIL.

It is also easy to see that with only ten randomly generated simulation computations, it is unlikely that we will generate worst-case computations in which K = 6. In fact, it is not clear that there exists a computation in which all processes issue all six message types for each CS cycle.

In summary, the Ricart-Agrawala algorithm performed both qualitatively and quantitatively as expected. The Maekawa algorithm performed qualitatively and very nearly quantitatively as expected.

V. FUTURE STUDIES

The results from these experiments show that even simple experiments can have unexpected results. I did not expect to see FAIL, YIELD, or INQUIRE messages when only one process at a time contended for the critical section. An interesting follow-up study would be to count the frequency at which these messages occur under different conditions.

This experiment used only the optimal quantized values of *N* from the billiard quorum algorithm. Billard quorums can be adjusted for use with other values of *N* by doing the following: generate quora for the quantized value of *N* greater than the desired *N*. Then delete all references to values greater than the desired *N*. Another experiment could measure message complexity in Maekawa for every integer value of *N* within a range, to see if the graph of message complexity vs. *N* forms a staircase function, with steps at the known quantization values of *N*.

Despite the simplicity and predictability of the Ricart-Agrawala algorithm, there are opportunities for further study of this algorithm as well. In their paper, they describe several situations in which a more specialized network could achieve an lower message complexity. For example, if a REPLY is not received within a certain time limit, then the lock is assumed to be granted.

References

[1] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Comm. ACM 21,* 7 (July 1978), pp. 558-564.

[2] G. Ricart and A.K. Agrawala, “An optimal algorithm for mutual exclusion in computer networks,” *Comm. ACM 24,* 1 (January, 1981), pp. 9-17.

[3] M. Maekawa, “A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems,” *ACM Trans. on Computer Systems* 3 (1985), pp. 145-159.

[4] B.A. Sanders, “The information structure of distributed mutual exclusion algorithms,” *ACM Trans. on Computer Systems 5*, 3 (1987), pp. 284-299.

[5] D. Agrawal, Ö Eğecuiğlu, A. Abbadi, “Billiard quorums on the grid*,” Info. Proc. Letters 64* (1997), pp. 9-16.

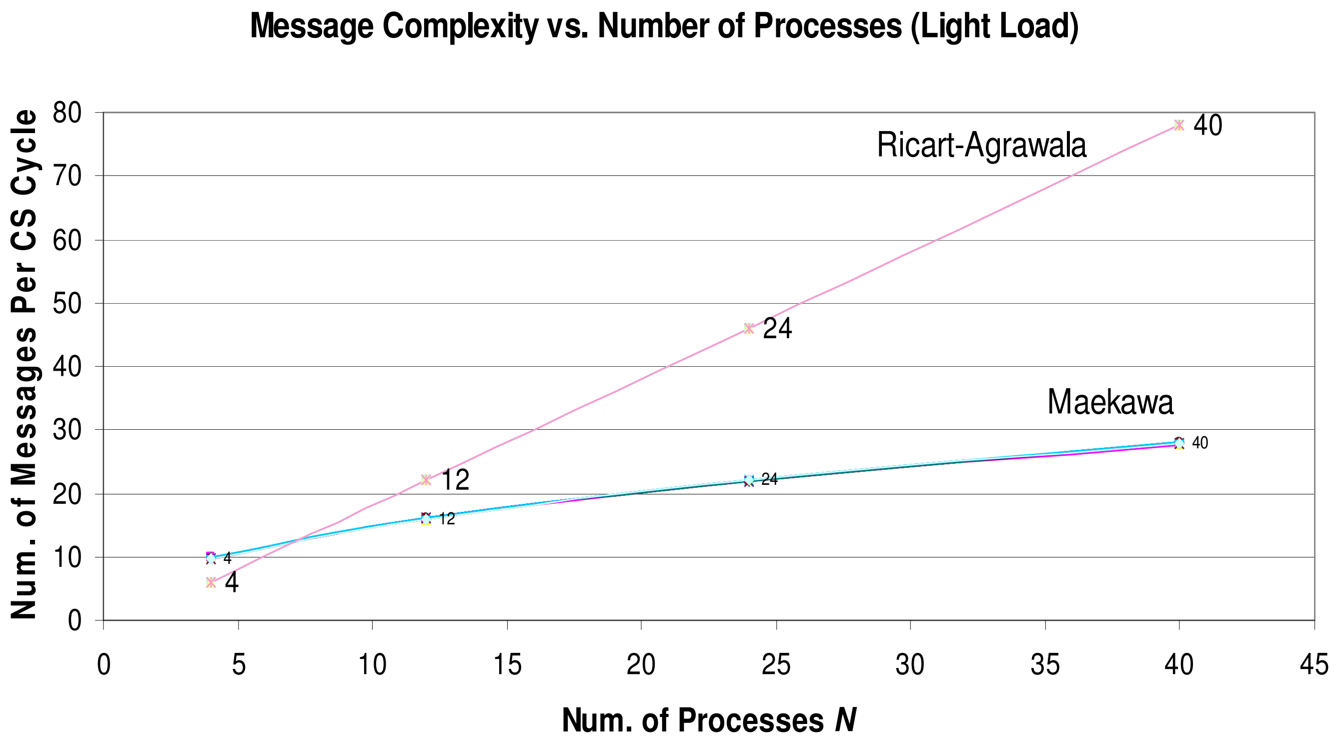

Fig. 3. Message Complexity vs. Number of Processes (light load)

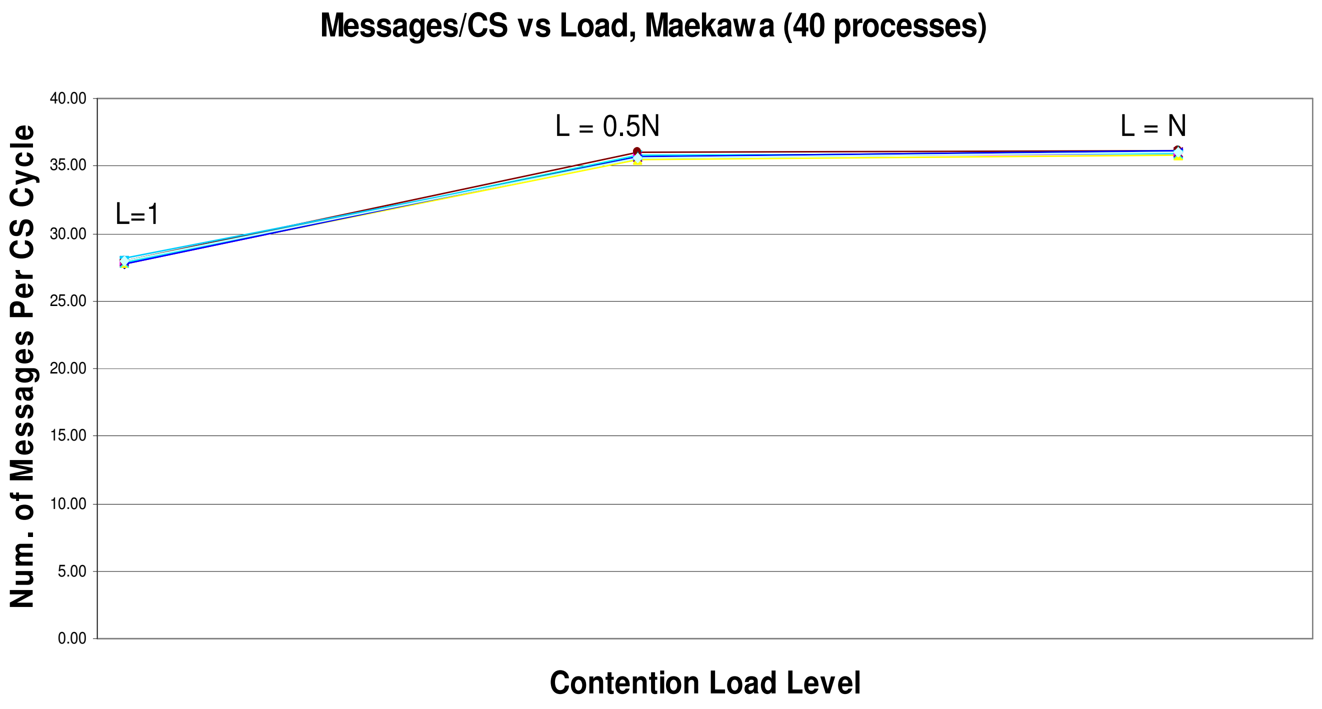

Fig. 4. Message Complexity vs. Load Level (40 processes)