Performance of Fair Distributed Mutual Exclusion Algorithms

Kandarp Jani and Ajay D. Kshemkalyani

Computer Science Department, Univ. of Illinois at Chicago, Chicago, IL 60607, USA
{kjani,ajayk}@cs.uic.edu

Abstract. The classical Ricart-Agrawala algorithm (RA) has long been considered the most efficient *fair* mutual exclusion algorithm in distributed message-passing systems. The algorithm requires $2(N-1)$ messages per critical section access, where N is the number of processes in the system. Recently, Lodha-Kshemkalyani proposed an improved *fair* algorithm (LK) that requires between N and $2(N-1)$ messages per critical section access, and without any extra overhead. The exact number of messages depends on the concurrency of requests, and is difficult to prove or analyze theoretically. This paper shows the superior performance of LK over RA using extensive simulations under a wide range of critical section access patterns and network loads.

1 Introduction

Mutual exclusion is a fundamental paradigm in computing. Over the past two decades, several algorithms have been proposed to achieve mutual exclusion in asynchronous distributed message-passing systems [2, 8]. Designing such algorithms becomes difficult when the the requirement for "*fair*" synchronization needs to be satisfied. The commonly accepted definition of fairness is that requests for access to the critical section (CS) are satisfied in the order of their logical timestamps [4]. If two requests have the same timestamp, the process identifier is used as a tie-breaker. Lamport's logical clock [4] is used to assign timestamps to messages to order the requests. The algorithm to update the clocks and to timestamp requests keeps all logical clocks closely sychronized. A fair mutual exclusion algorithm needs to guarantee that requests are accessed in increasing order of the timestamps. Of the many distributed mutual exclusion algorithms, the only algorithms that are fair in the above context are Lamport [4], Ricart-Agrawala (RA) [7], and Lodha-Kshemkalyani (LK) [5].

The performance metrics for mutual exclusion algorithms are the following: the *number of messages*, the *synchronization delay*, the *response time*, and the *waiting time*. Others such as the *throughput* can be expressed in terms of the above metrics and inherent characteristics of the programs – such as the CS execution time, and the time spent executing non-CS code. For a system consisting of N processes, let d denote the time for a message hop and css, the time spent executing the CS. The lower bounds on the *waiting time* T, the *response*

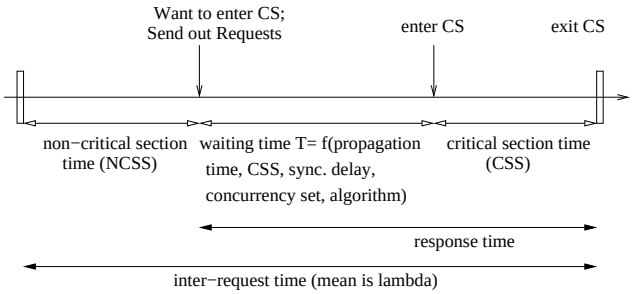

Fig. 1. The relationships among CSS, NCSS, λ, and Waiting time at a process.

time (which is $T + css$), and the *synchronization delay* are $2d$, $2d + css$, and d, respectively, for both the Lamport [4] and the RA [7] algorithms. The message complexity of Lamport is $3(N-1)$ messages per CS access. The message complexity of RA is $2(N-1)$ messages per CS access. The RA algorithm has been considered a classic in mutual exclusion algorithms, since 1983. It has been presented in many textbooks on distributed algorithms and distributed systems.

Recently, Lodha and Kshemkalyani proposed an improved algorithm (LK) [5], over the RA algorithm. This algorithm uses the same model as RA, but has a message complexity between N and $2(N-1)$ messages per CS access. All other metrics measure the same as or better than those of RA. The exact number of messages per CS access depends on the concurrency of the requests being made.

The improvement achieved by LK is difficult to determine theoretically or analytically. In this paper, we show via simulations that the performance of LK is better than that of RA. We consider two performance measures – the number of messages and the waiting time, to access the CS. As the response time can be derived from the waiting time, we view the waiting time as a more fundamental measure than the response time. The simulations study the performance of the algorithms under a wide range of requesting patterns (high load and low load), and under a wide range of program behaviors (time spent executing the CS). The study also accounts for a wide range of network dimensions and loads.

2 Overview of RA and LK Algorithms

The system model assumes there are N processes in the error-free asynchronous message-passing system. Message transit times may vary. Channels are FIFO. It is assumed that a single process runs on each site. So a site is synonymous with a process. A process requests a CS by sending REQUEST messages and waits for appropriate REPLY messages from the other processes. While a process is waiting to enter the CS, it cannot make another request to enter CS. Each process executes the following loop forever: execute the noncritical section, request to enter the CS, wait to get permission, execute the CS. In each iteration, the three durations are denoted NCSS, Waiting, and CSS. The time relations among the durations, and λ, the mean inter-request time, are shown in Fig. 1.

2.1 Ricart-Agrawala's Algorithm [7]

1. When a process P_i wants to enter the CS, it sends a timestamped REQUEST message to the other processes.
2. When a process P_j receives a REQUEST from P_i, it sends a REPLY to P_i if (i) P_j is not requesting nor executing the CS, or (ii) P_j is requesting with lower priority. Otherwise, P_j defers P_i's request.
3. P_i can enter the CS after it receives a REPLY from all other processes.
4. When P_i exits its CS, it sends as REPLY to all the processes whose requests it had deferred (step 2).

Thus, there are exactly $2(N-1)$ messages exchanged per CS access.

2.2 Lodha-Kshemkalyani Algorithm [5]

The LK algorithm assumes the same system model as that of RA but reduces the number of messages required per CS access. To realize this objective, LK uses three types of messages and a queue, the Local Request Queue (LRQ), which contains "concurrent requests".

Concurrent Requests: Consider two requests R_i initiated by process P_i and R_j initiated by process P_j. R_i and R_j are concurrent iff P_i's REQUEST is received by P_j after P_j has made its REQUEST and vice versa.

Concurrency Set: The concurrency set of request R_i^k, the k^{th} request made by P_i, is defined as: $CSet_i(R_i^k) = \{R_j | R_i \textit{ is concurrent with } R_j\} \bigcup \{R_i\}$. As a single request by P_i can be outstanding at a time, we simply use $CSet_i$ to denote its concurrency set.

Three types of messages are used by the LK algorithm: REQUEST, REPLY, and FLUSH. The REQUEST message contains the timestamp of the request. The REPLY and FLUSH messages contain the timestamp of the last completed CS access by the sender of that REPLY or FLUSH message. The REQUEST and REPLY messages hold a different significance from that in the RA algorithm, and have substantially enhanced semantics! FLUSH is the extra type used by the LK algorithm to achieve the savings in the messages. We emphasize that the size of messages used by LK is the same as that used by RA. The savings in the number of messages is not at the cost of any other parameter. The following observations indicate how the savings are achieved.

Observations

1. All requests are totally ordered by priority, similar to the RA algorithm.
2. A process receiving a REQUEST message can immediately determine whether the requesting process or itself should be allowed to enter the CS first.
3. Multiple uses of the REPLY message:
   - It acts a reply from a process that is not requesting.
   - It acts a collective reply from processes with higher priority requests.

The $REPLY(R_j)$ message from P_j indicates that R_j is the latest REQUEST for which P_j executed the CS. This indicates that all requests that have priority greater than that of R_j have finished CS and are no longer in contention. When a process P_i receives $REPLY(R_j)$, it can remove those REQUESTs whose priority $\geq$ priority of R_j, from its local request queue (LRQ_i). Thus, $REPLY(R_j)$ is a logical reply that denotes a collective reply from all processes that had made higher priority requests than or equal to R_j.

4. Multiple uses of FLUSH message: A FLUSH message is sent by a process after executing the CS, to the concurrently requesting process with the next highest priority (if it exists.) When entering the CS, a process can determine the state of all other processes in some consistent state with itself. Any other process is either requesting CS access and its (lower) priority is known, or it is not requesting. After executing CS, P_i sends a $FLUSH(R_i)$ message to P_j which is the concurrently requesting process with the next highest priority. $FLUSH(R_i)$ is a logical reply that denotes a collective reply from all processes that had made higher priority requests than or equal to R_i.
5. Multiple uses of REQUEST message: A process P_i that wants to invoke CS sends a REQUEST message to all other processes. On receipt of a REQUEST message, a process P_j that is not requesting sends a REPLY message immediately. If process P_j is requesting concurrently, it does not send a REPLY message. If P_j's REQUEST has a higher priority, the received REQUEST from P_i serves a reply to P_j. P_j will eventually execute CS (before P_i) and then through a chain of FLUSH/REPLY messages, P_i will eventually receive a logical reply to its REQUEST. If P_j's REQUEST has a lower priority than P_i's REQUEST, P_j likewise awaits P_i's logical permission via a chain of FLUSH/REPLY messages.

RA-Type Messages: The REPLY messages sent by concurrently requesting processes in RA, but not in LK (where LRQ prioritizes concurrent requests).

3 Objectives of Simulation

The total number of messages used for a particular CS access is $2N - |Cset|$, where $Cset$ is the concurrency set of that CS access request [5]. This is because there are $N-1$ REQUEST messages, $(N-1) - |Cset|$ REPLY messages, and 1 FLUSH message. The number of concurrent requests potentially depends on: the number of processes, inter-request time, time spent in the CS, and the propagation delay. The actual number of messages in a real system is difficult to analyze theoretically. The objectives of the simulation are as follows.

- To measure the message overhead of LK, per CS access, under a wide range of requesting conditions and network conditions. The message overhead of RA is always $2(N-1)$ messages per CS access.
- To compare the waiting time of RA and LK under varying requesting and network conditions.

3.1 Simulation Parameters

Input Parameters

1. **Number of processes (N):** As N increases, and assuming that the mean inter-request time is not changed, there are more requests for the CS. This affects the concurrency set and waiting time also increases. Hence N is an important parameter. By varying N, we also study scalability. On the Intel Pentium 3 with 128 MB RAM used for the simulation, up to 45 processes could be simulated. Note that the earlier comprehensive performance study of distributed mutual exclusion algorithms assumed only 21 processes, and did not test for sensitivity to N [2].
2. **Inter-request Time (λ):** Inter-request time is the time between generating two requests by a process. This parameter is exponentially distributed with λ as the mean. As processes begin requesting more furiously (λ decreases), there will be more requests, the probability of concurrent requests is higher, and hence a reduction in the number of messages per CS access. λ directly affects the concurrency set. Also, the inter-request time is related to the waiting time, propagation delay and CSS (see Section 3.2), and is therefore of interest. The typical values of the mean λ used in the simulations range from 10^{-4}s to 10s.
3. **Critical Section Sitting Time (CSS):** The critical section sitting time is the amount of time a process executes in the critical section. It is modeled as an exponential distribution with a mean of CSS. It is difficult to analyze how the concurrency set is affected by CSS. However, CSS impacts the waiting time (see Section 3.2). Also, a process cannot request when executing the CS. This puts a bound on how frequently a process can request the CS. The values of the mean CSS used in the simulations range from 10^{-7}s to 10^{-3}s.
4. **Propagation Delay (D):** The link propagation delay is the time elapsed while propagating a message from one process to another over the network. Realistic systems inherently exhibit this delay, and the waiting time at a process depends on this delay. The network is a complex entity to model [3]. As we would like to consider a single parameter to characterize (i) physical network size and/or distances, (ii) speed for all the links, and (iii) congestion, we model transmission time as an exponential distribution about the mean, D, as representative of all links. This distribution can also approximate TCP delays [1]. While simulating the mutual exclusion algorithms, we only implicitly model this parameter D because, (i) it follows the same distribution as CSS, and (ii) the occurrence of this delay D is tightly coupled with the CSS (see Section 3.2). Rather, we assume that CSS implicitly includes D. Note that the earlier comprehensive performance study of distributed mutual exclusion algorithms did not model this delay [2].

Output Parameters

1. **Normalized Message Complexity (M_{norm}):** The normalized message complexity is ((total number of messages exchanged per CS access) / N).
2. **Waiting Time (T):** The waiting time is the time a process has to wait to enter the CS after requesting the CS. The LK algorithm uses LRQ to

track the concurrent requests at each process. By having to wait for fewer replies, LK's waiting time decreases with respect to RA but enqueuing and dequeuing may add some time overhead.

3.2 Inter-relation amongst CSS, λ, T, and D

- The mean inter-request time λ equals the mean critical section sitting time (CSS), the mean waiting time (T), and the mean noncritical section time ($NCSS$), as seen from Fig. 1. Further, T is a function of the CSS, D, the concurrency set, and the mutual exclusion algorithm used. Thus, $\lambda = CSS + NCSS + T = CSS + NCSS + f(CSS, D, Cset, ME.\ algorithm)$ λ, CSS, and D are assumed to be the means of exponential distributions. Hence, propagation time can be viewed as being incorporated in CSS.
- $\lambda > CSS$ because a process cannot request when executing the critical section and the rate of CS executions cannot exceed the rate of request generation. If we allow the processes to request while executing the critical section, those requests will be lost. As a result, the input distribution of CS requests will no longer remain exponential.
- The total waiting time of the system is directly proportional to CSS and D. As CSS increases, system-wide waiting time also increases. The average-case waiting time $T_{avg} = (|CSet|/2)(CSS + D)$. This equation also justifies why the propagation delay D can be viewed as being incorporated within CSS.

The above points explain how the value of λ is constrained by CSS, D, and T.

4 Simulation Results

4.1 Experimental Setup

The LK and RA algorithms were implemented in C using the simulation framework of OPNET [6]. We report three experiments, in which we test the performance for various combinations of the input parameters N, CSS, λ.

1. The number of messages exchanged in the system was measured for multiple settings of the tuple (N, CSS), as the mean inter-request time (λ) is varied.
2. The number of messages exchanged in the system was measured for multiple settings of the tuple (CSS, λ), as the number of processes (N) increased.
3. The average waiting time (T) in the system was measured for both the LK and RA algorithms, for different settings of the tuple (CSS, λ), as the number of processes (N) was varied.

The machine used for simulation is an Intel Pentium 3 with 128 MB of RAM. For each simulation run, statistics were collected for 1000 CS requests per process, which amounted to a minimum of 10,000 requests and a maximum of 45,000 requests. For each simulation run, the statistics collected for the initial 10% requests were discarded to eliminate the effects of startup. Each statistic reported in the results is an average of the statistics of ten simulation runs with different seeds. Propagation delay (D) was implicitly accounted for (Section 3).

The ranges and distributions of the three input parameters were discussed in Section 3.1. Based on the observations (Section 3.2), the range of λ is adjusted based on the value of CSS (and D).

To present the results on message overhead, following statistics are collected.

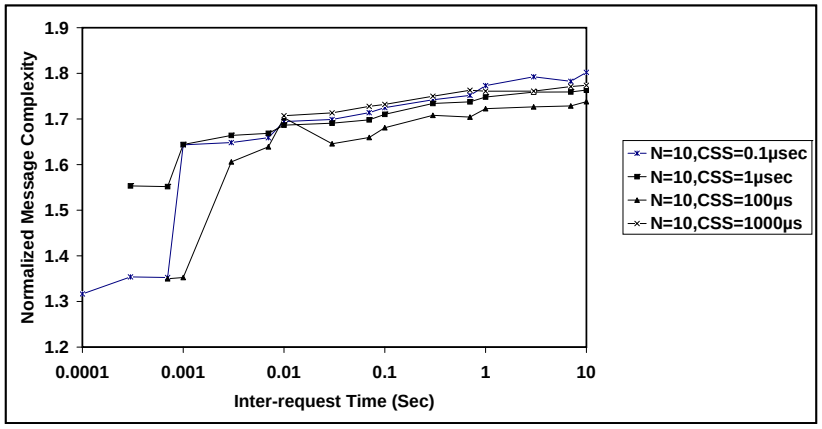

Fig. 2. Normalized message complexity vs. inter-request time (SP_{1x}).

1. Total number of requests messages in the system after steady state. (M_{req})
2. Total number of messages in the system after steady state. (M_{tot})

For each experiment, the normalized message complexity is reported as $M_{norm} = M_{tot}/M_{req}$. For RA, $M_{norm} = 2$. M_{norm} for KS will vary from 1 to 2.

4.2 Expt. 1: Impact of Inter-request Time (λ) on Message Overhead

The worst-case message complexity of the LK algorithm is the same as that of RA. However, on an average, LK performs better than RA. Here, the goal is to study LK's message overhead M_{norm} as the processes request more furiously. The simulations were performed for 20 settings of the tuple (N, CSS) while varying λ from 10^{-4} to 10^{1}. The settings were formed by taking all combinations of the values of N: 10, 20, 30, and 40, and the 4 values of CSS: 10^{-7}, 10^{-6}, 10^{-4}, and 10^{-3}. The results are plotted in Figs. 2, 3, 4, and 5.

Fig. 2: $SP_{11}(10, 10^{-7})$, $SP_{12}(10, 10^{-6})$, $SP_{13}(10, 10^{-4})$, $SP_{14}(10, 10^{-3})$
Fig. 3: $SP_{21}(20, 10^{-7})$, $SP_{22}(20, 10^{-6})$, $SP_{23}(20, 10^{-4})$, $SP_{24}(20, 10^{-3})$
Fig. 4: $SP_{31}(30, 10^{-7})$, $SP_{32}(30, 10^{-6})$, $SP_{33}(30, 10^{-4})$, $SP_{34}(30, 10^{-3})$
Fig. 5: $SP_{41}(40, 10^{-7})$, $SP_{42}(40, 10^{-6})$, $SP_{43}(40, 10^{-4})$, $SP_{44}(40, 10^{-3})$

The mean inter-request time (λ) is varied as per the constraint imposed (see Section 3.2). As seen from the plots for sets $SP_{x3}(N, 10^{-4})$ and $SP_{x4}(N, 10^{-3})$, there are no data points for $\lambda < 7 \times 10^{-4}$ and $\lambda < 7 \times 10^{-3}$ respectively.

Observations

- As the value of λ increases, the value of M_{norm} increases but is still lower than the value for RA, which is 2. With an increase in the value of λ, the concurrency set grows sparse. This results in more number of messages of RA-type being exchanged. Thus, LK performs much better when the load on the system is heavier, conforming to the expression, $2N - |Cset|$, for the message overhead.

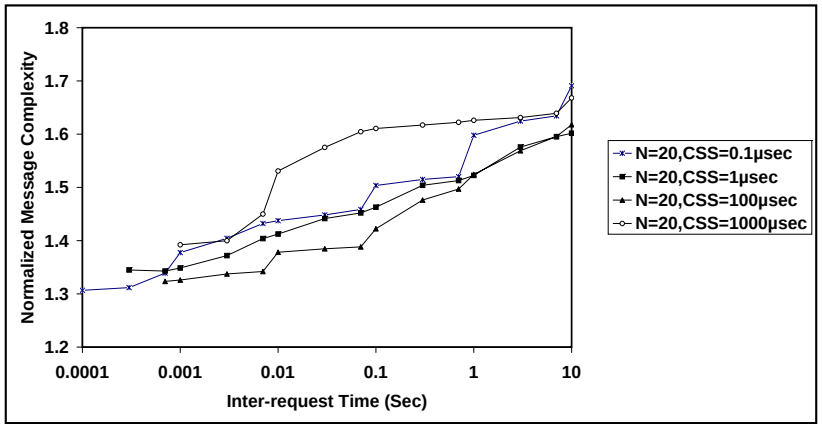

Fig. 3. Normalized message complexity vs. inter-request time (SP_{2x}).

- For lower values of N (Fig. 2), as λ increases, there is a jump in the value of M_{norm} which later saturates, and tends towards a value of 1.8. But for higher N (Figs. 3, 4, 5), M_{norm} is lower and follows a smoother curve. The effect of N on M_{norm} is studied in Section 4.3.
- No definite relationship can be readily inferred between CSS and M_{norm}.

Thus, we experimentally see how the LK algorithm outperforms RA under heavy CS request load. Even under light load, LK shows some improvement over RA.

4.3 Expt. 2: Scalability with Increasing Number of Processes

This experiment studies the message overhead of the LK algorithm as the number of processes in the system is increased. This also measures scalability. The simulations were performed for 20 settings of the tuple (CSS, λ) while varying N from 10 to 45. The settings were formed by taking all combinations of the 4 values of CSS: 10^{-7}, 10^{-6}, 10^{-4}, and 10^{-3}, and the 5 values of λ: 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1} and 10^{1}. The results are plotted in Figs. 6, 7, 8, and 9.

Fig. 6: $SC_{11}(10^{-7}, 10^{-4})$, $SC_{12}(10^{-7}, 10^{-3})$, $SC_{13}(10^{-7}, 10^{-2})$, $SC_{14}(10^{-7}, 10^{-1})$, $SC_{15}(10^{-7}, 1)$
Fig. 7: $SC_{21}(10^{-6}, 10^{-4})$, $SC_{22}(10^{-6}, 10^{-3})$, $SC_{23}(10^{-6}, 10^{-2})$, $SC_{24}(10^{-6}, 10^{-1})$, $SC_{25}(10^{-6}, 1)$
Fig. 8: $SC_{31}(10^{-4}, 7{\times}10^{-4})$, $SC_{32}(10^{-4}, 10^{-3})$, $SC_{33}(10^{-4}, 10^{-2})$, $SC_{34}(10^{-4}, 10^{-1})$, $SC_{35}(10^{-4}, 1)$
Fig. 9: $SC_{41}(10^{-3}, 7{\times}10^{-3})$, $SC_{42}(10^{-3}, 10^{-2})$, $SC_{43}(10^{-3}, 10^{-1})$, $SC_{44}(10^{-3}, 1)$

Note from Fig. 9 that SC_{4x} consists of four data sets instead of five as $\lambda > CSS$.

Observations

- As N increases, M_{norm} decreases first but then levels off. This shows the scalability of LK. With an increase in N and at a fixed λ, the number of

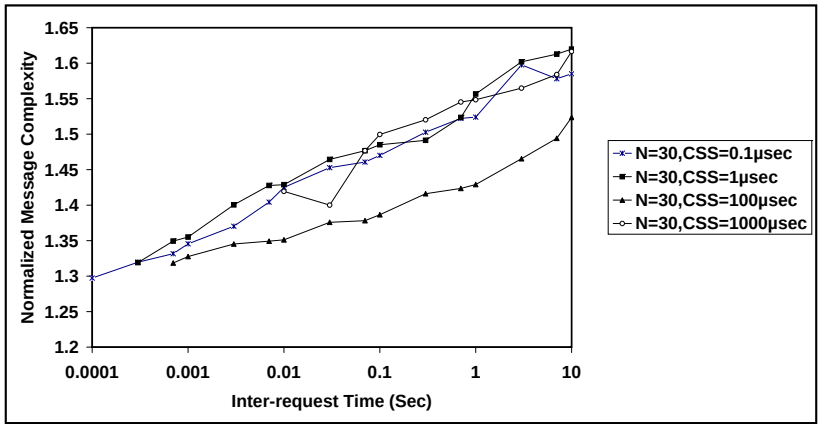

Fig. 4. Normalized message complexity vs. inter-request time (SP_{3x}).

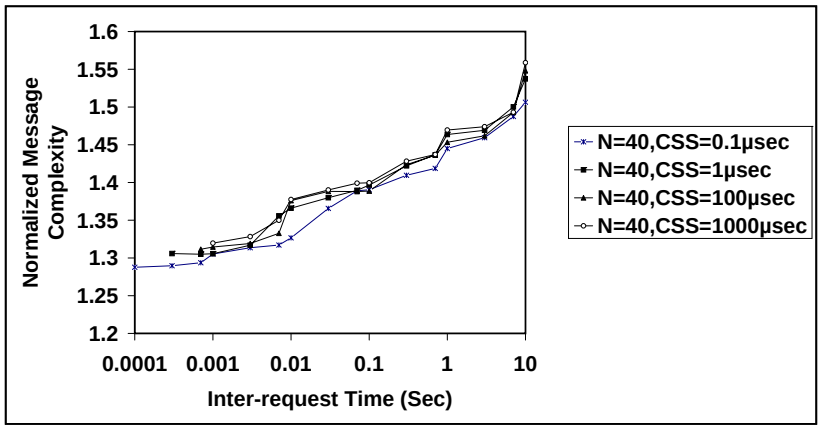

Fig. 5. Normalized message complexity vs. inter-request time (SP_{4x}).

concurrent requests increases (load increases), and hence the waiting time increases. Waiting times will tend to overlap more, as also the non-CSS reduces. This potentially affects the probability of two processes making concurrent requests. The exact impact observed on M_{norm} is difficult to explain by theory.

- For low values of N, as N increases, the dip in M_{norm} is quite noticable. However, the curves tend to saturate for $N > 30$. This suggests that M_{norm} will tend to a steady value as the number of processes increases.
- Another observation which complements the results of Section 4.2 is that, for lower values of λ, the curves are also lower. Lesser the inter-request time, the probability that there will be more number of concurrent requests increases. Consequently the message overhead reduces. As the value of λ increases, so does the normalized message complexity.

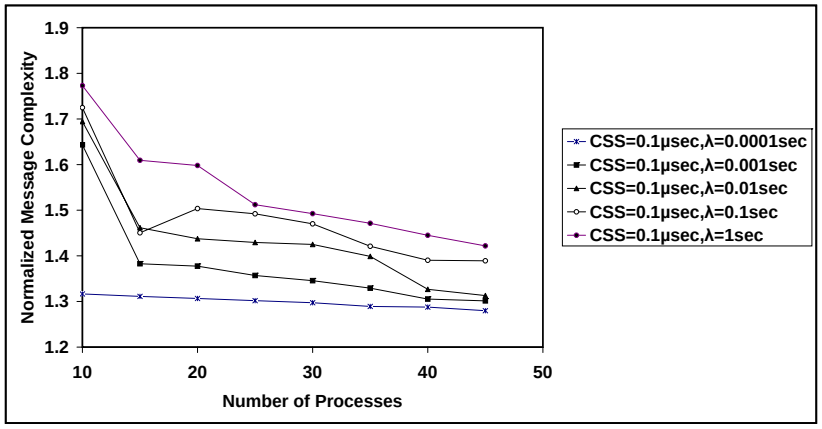

Fig. 6. Normalized message complexity vs. number of processes (SC_{1x}).

Recall from Section 4.2 the propagation delay D is treated as being a part of CSS. Modeling D independently is beyond the scope of this simulation.

4.4 Expt. 3: Improvement in Waiting Time

This experiment compares the waiting times in the RA and LK algorithms. Theoretically, it can be predicted that the waiting time of LK is at least as good as that of RA. This is because the "RA-type" messages are absent in LK, and hence, a requesting process need not wait to receive REPLY messages from all the concurrently requesting processes. Specifically, LK has two cases.

- A high-priority process does not have to wait for a REPLY message from a lower priority process. Under low load, this savings may not be large because there are not many processes requesting concurrently; thus, there may not be many lower priority concurrent requests. Under high load, this savings may not be large because there are many concurrently requesting processes, and a process may need to wait anyway because there are several higher priority processes that need to execute their CS before this process can enter its CS. Hence, not having to wait for the REPLY messages from lower priority processes may not reduce the waiting time substantially.
- A low-priority process does not have to wait for a REPLY message from all the higher priority processes. It suffices if the FLUSH/REPLY message with the *immediately* higher priority than that of the requesting process, reaches the requesting process; the LRQ queue would get purged of all higher priority requests than that on the received REPLY/FLUSH. Here, the reduction in waiting time may not be high, as also seen from the following scenario. Assume that P_j's request has a higher priority than P_k's request which has a higher priority than P_i's request. Statistically, the probability of a REPLY/FLUSH sent later by P_k arriving at P_i earlier than the REPLY of P_j (which would be sent in RA) is low.

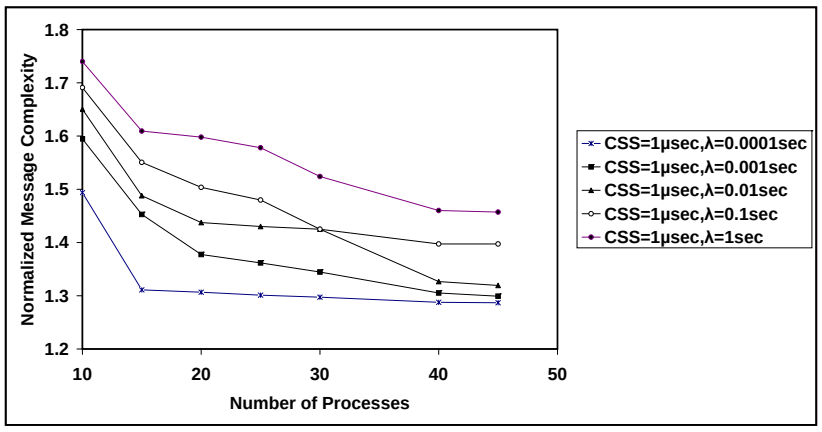

Fig. 7. Normalized message complexity vs. number of processes (SC_{2x}).

It is difficult to theoretically analyze the improvement in waiting time of LK over RA. Hence, we simulate both algorithms. The curves in Figs. 10 and 11 are plotted for the following settings of (CSS, λ), with N varying from 10 to 45.
Fig. 10: $WT_1(10^{-6}, 10^{-4})$
Fig. 11: $WT_2(10^{-7}, 10^{-4})$
In the graphs, the RA curve is clearly above the LK curve. Thus, LK gives a better waiting time than RA. The following observations can also be made.

Observations

- Initially, with a low N, the concurrency set is small. The LK curve follows the RA curve but is below it because of having to wait for fewer messages.
- As the value of N increases, contention for CS increases. The rate of increase of waiting time for RA curve remains the same. However, the same decreases for the LK curve. This behavior can be attributed to the increase in the size of the concurrency set with increasing N. A larger concurrency set implies not having to wait for more number of RA-type replies (that will never get sent in the LK algorithm). Consequently, there is a relative reduction in the waiting time for LK.
- As N is increased further, however, the two curves start getting closer. The LK curve runs much more closer to the RA curve for $N > 30$. This may be attributed to the fact that there are now more higher-priority processes that need to execute their CS first. Therefore, not having to wait for the REPLY messages from lower priority processes as well as (earlier) higher priority processes is not as effective in reducing the waiting time. Another reason is that, along with increase in the size of the concurrency set, the number of enqueue and dequeue operations (for LRQ) at each process also increases. and their overhead becomes nontrivial.

The simulations show that the waiting time in the LK algorithm is somewhat lower than in the RA algorithm under all conditions tested.

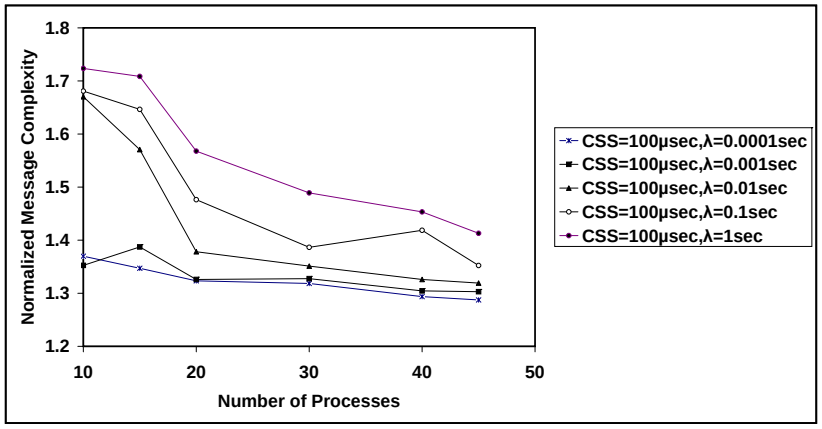

Fig. 8. Normalized message complexity vs. number of processes (SC_{3x}).

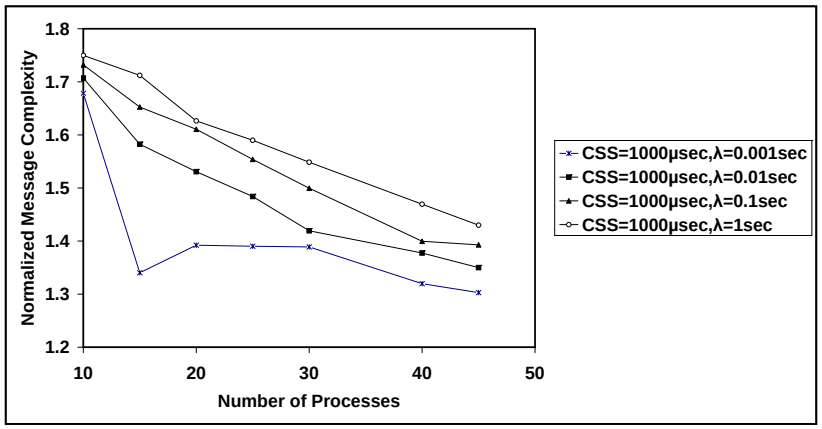

Fig. 9. Normalized message complexity vs. number of processes (SC_{4x}).

5 Conclusions

The RA algorithm was the most efficient *fair* algorithm for distributed mutual exclusion for about two decades. This paper experimentally studied the performance of the recently proposed LK algorithm, and showed that it outperforms the RA algorithm in both message complexity and waiting time, without compromising on *fairness* or any other metrics.

Acknowledgements. This work was supported by US NSF grant CCR-9875617. We thank Shashank Khanvilkar for his help with OPNET.

References

1. P. Chandra, P. Gambhire, A. D. Kshemkalyani, Performance of the Optimal Causal Multicast Algorithm: A Statistical Analysis, IEEE Transactions on Parallel and Distributed Systems, 15(1):40-52, January 2004.

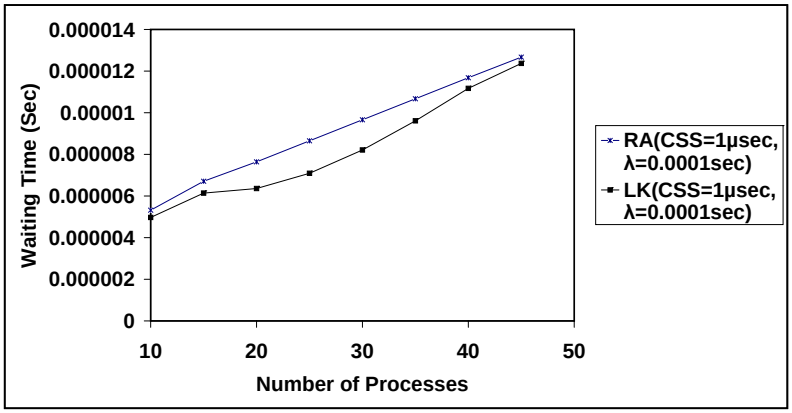

Fig. 10. Waiting time vs. number of processes (WT_1).

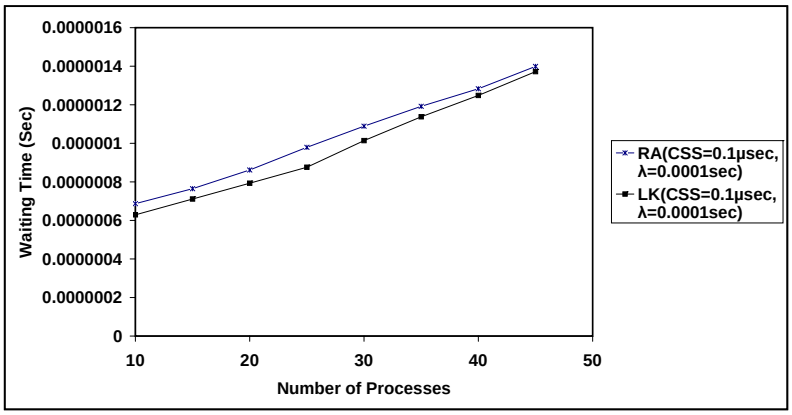

Fig. 11. Waiting time vs. number of processes (WT_2).

2. Y.-I. Chang, A Simulation Study on Distributed Mutual Exclusion, J. Parallel and Distributed Computing, Vol. 33(2): 107-121, 1996.
3. S. Floyd, V. Paxson, Difficulties in Simulating the Internet, IEEE/ACM Transactions on Networking, Vol. 9(4): 392-403, August 2001.
4. L. Lamport, Time, Clocks and the Ordering of Events in Distributed Systems, Comm. ACM, Vol. 21(7): 558 - 565, Jan 1978.
5. S. Lodha, A. Kshemkalyani, A Fair Distributed Mutual Exclusion Algorithm, IEEE Trans. on Parallel and Distributed Systems, 11(6): 537-549, June 2000.
6. OPNET, Available at: ⟨http://www.opnet.com/products/modeler/home.html⟩
7. G. Ricart, A. K. Agrawala, An Optimal Algorithm for Mutual Exclusion in Computer Networks, Comm. ACM, 24(1):9-17, Jan. 1981.
8. M. Singhal, A Taxonomy of Distributed Mutual Exclusion, J. Parallel and Distributed Computing, 18(1):94-101, May 1993.