A Fair Distributed Mutual Exclusion Algorithm

Sandeep Lodha and Ajay Kshemkalyani, *Senior Member*, *IEEE*

Abstract—This paper presents a fair decentralized mutual exclusion algorithm for distributed systems in which processes communicate by asynchronous message passing. The algorithm requires between $N-1$ and $2(N-1)$ messages per critical section access, where N is the number of processes in the system. The exact message complexity can be expressed as a deterministic function of concurrency in the computation. The algorithm does not introduce any other overheads over Lamport's and Ricart-Agrawala's algorithms, which require $3(N-1)$ and $2(N-1)$ messages, respectively, per critical section access and are the only other decentralized algorithms that allow mutual exclusion access in the order of the timestamps of requests.

Index Terms—Algorithm, concurrency, distributed system, fairness, mutual exclusion, synchronization.

1 Introduction

The mutual exclusion problem states that only a single process can be allowed access to a protected resource, also termed as a critical section (CS), at any time. Mutual exclusion is a form of synchronization and is one of the most fundamental paradigms in computing systems. Mutual exclusion has been widely studied in distributed systems where processes communicate by asynchronous message passing, and a comprehensive survey is given in [2], [9]. For a system with N processes, competitive algorithms have a message complexity between $log(N)$ and $3(N-1)$ messages per access to the CS, depending on their features. Distributed mutual exclusion algorithms are either token-based or nontoken-based. In token-based mutual exclusion algorithms, a unique token exists in the system and only the holder of the token can access the protected resource. Examples of token-based mutual exclusion algorithms are Suzuki-Kasami's algorithm [12] (N messages), Singhal's heuristic algorithm [11] ($[N/2, N]$ messages), Raymond's tree-based algorithm [6] ($log(N)$ messages), Yan et al.'s algorithm [13] ($O(N)$ messages), and Naimi et al.'s algorithm [5] ($O(log(N))$ messages). Nontoken-based mutual exclusion algorithms exchange messages to determine which process can access the CS next. Examples of nontoken-based mutual exclusion algorithms are Lamport's algorithm [3] ($3(N-1)$ messages), Ricart-Agrawala's algorithm [7] ($2(N-1)$ messages), Carvalho-Roucairol's variant of the Ricart-Agrawala algorithm [1] ($[0, 2(N-1)]$ messages), Maekawa's algorithm [4] ($[3\sqrt{N}, 5\sqrt{N}]$ messages), and Singhal's dynamic information structure algorithm [10] ($[N-1, 3(N-1)/2]$ messages). Sanders gave a theory of information structures to design mutual exclusion algorithms, where an information structure describes which processes maintain information about what other processes, and from which processes a process must request information before entering the CS [8].

Due to the absence of global time in a distributed system, timestamps are assigned to messages according to Lamport's clocks [3]. In the context of mutual exclusion, Lamport's clocks are operated as follows: Each process maintains a scalar clock with an initial value of 0. Each time a process wants to access the CS, it assigns that request a timestamp which is one more than the value of the clock. The process sends the timestamped request to other processes to determine whether it can access the CS. Each time a process receives a timestamped request from another process seeking permission to access the CS, the process updates its clock to the maximum of its current value and the timestamp of the request.

Fairness is a very important criteria for solutions to most real-life resource contention problems. The commonly accepted definition of fairness in the context of mutual exclusion is that requests for access to the CS are satisfied in the order of their timestamps. Of all the distributed mutual exclusion algorithms in the literature, only the nontoken-based algorithms of Lamport [3] and Ricart-Agrawala [7] (RA) are fair in the sense described above. Singhal's heuristic algorithm [11] guarantees some degree of fairness but is not fair in the sense described above. A lower priority request can execute CS before a higher priority request if the higher priority request is delayed. The algorithm has a different criteria for fairness. It favors sites which have executed their CSs least frequently and discourages sites which have executed CSs heavily. This does not take into account the causality relation that exists between two requests, and hence, does not conform to the sense of fairness described by Lamport's clock. Singhal's dynamic

• *S. Lodha is with Synposys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043. Email: lodha@synopsys.com*
• *A. Kshemkalyani is with the Department of Electrical Engineering and Computer Science (MC 154), 1120 Science and Engineering Offices, 851 S. Morgan St., University of Illinois at Chicago, Chicago, IL 60607-7053. E-mail: ajayk@eecs.uic.edu.*

Manuscript received 22 July 1998; revised 4 Nov. 1999; accepted 6 Jan. 2000. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107178.

information structure algorithm [10] attempts to be fair, but does not satisfy the fairness criterion. The algorithm uses the concept of Lamport's clock and the causality relationship, but it also allows a low priority request to execute CS before a high priority request if the high priority request is on the way or delayed (process that has made a higher priority request is not in the request set of the process that has made the low priority request). The proposed algorithm in this paper uses the fairness criteria given by Lamport and improves on RA, which is the best known algorithm that guarantees fairness in the same sense.

Lamport's fair mutual exclusion algorithm requires $3(N-1)$ messages per CS access. Ricart-Agrawala's fair mutual exclusion algorithm optimizes Lamport's algorithm and requires $2(N-1)$ messages per CS access. In this paper, we present a fair mutual exclusion algorithm that requires between $N-1$ and $2(N-1)$ messages per CS access. The exact number of messages for any CS access is $2(N-1)-x$, where x is the number of other requests that are made concurrently with this request. The presented algorithm uses the same system model as in the Lamport and Ricart-Agrawala algorithms and does not introduce any overheads. Mutual exclusion in shared memory systems is a very different problem and we do not address it here [14], [15].

Section 2 describes the system model and reviews the Ricart-Agrawala algorithm. Section 3 presents the new algorithm. Section 4 proves that the algorithm guarantees mutual exclusion and progress and is fair. This section also analyzes the message complexity. Section 5 gives concluding remarks.

2 Preliminaries

In this section, we describe the general system model and review the Ricart-Agrawala (RA) algorithm which is the best known fair distributed mutual exclusion algorithm [7]. The algorithm proposed in Section 3 is an improvement over the RA algorithm.

2.1 System Model

The RA algorithm and the algorithm by Lamport assume the following model. There are N processes in the system. The processes communicate only by asynchronous message passing over an underlying communication network which is error-free and over which message transit times may vary. Processes are assumed to operate correctly. Unlike the RA algorithm but similar to Lamport's algorithm, we assume FIFO channels in the communication network. Without loss of generality, we assume that a single process executes at a site or a node in the network system graph. Hence, the terms process, site, and node are interchangeably used.

A process requests a CS by sending REQUEST messages and waits for appropriate replies before entering its CS. While a process is waiting to enter its CS, it cannot make another request to enter another CS. Each REQUEST for CS access is assigned a priority and REQUESTs for CS access should be granted in order of decreasing priority for fair mutual exclusion. The priority or identifier, $ReqID$, of a request is defined as $ReqID$ = (SequenceNumber, PID), where SequenceNumber is a unique locally assigned sequence number to the request and PID is the process identifier. SequenceNumber is determined as follows. Each process maintains the highest sequence number seen so far in a local variable $Highest\_Sequence\_Number\_Seen$. When a process makes a request, it uses a sequence number which is one more than the value of $Highest\_Sequence\_Number\_Seen$. When a REQUEST is received, $Highest\_Sequence\_Number\_Seen$ is updated as follows:

$$\begin{aligned}&Highest\_Sequence\_Number\_Seen\\&\quad= \text{maximum}(Highest\_Sequence\_Number\_Seen,\\&\qquad \text{sequence number in the REQUEST}).\end{aligned}$$

Priorities of two REQUESTs, $ReqID_1$ and $ReqID_2$, where $ReqID_1 = (SN_1, PID_1)$ and $ReqID_2 = (SN_2, PID_2)$, are compared as follows. Priority of $ReqID_1$ is greater than priority of $ReqID_2$ iff $SN_1 < SN_2$ or ($SN_1 = SN_2$ and $PID_1 < PID_2$). All REQUESTs are thus totally ordered by priority. This scheme implements a variant of Lamport's clock mentioned in Section 1, and when requests are satisfied in the order of decreasing priority, fairness is seen to be achieved.

2.2 Review of Ricart-Agrawala Algorithm

The algorithm uses two types of messages: REQUEST and REPLY.

2.2.1 *Data Structure for Process P_i*

Each process P_i uses the following local integer variables: $My\_Sequence\_Number_i$, $ReplyCount_i$, and $Highest\_Sequence\_Number\_Seen_i$. P_i also uses the following vector:

- $RD_i[1:N]$ of Boolean. $RD_i[j]$ indicates if P_i has deferred the REQUEST sent by P_j.

2.2.2 *Algorithm*

The RA algorithm is outlined in Fig. 1. Each procedure in the algorithm is executed atomically.

The REPLY messages sent by a process are blocked only by processes that are requesting the CS with higher priority. Thus, when a process sends REPLY messages to all deferred requests, the process with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their decreasing priority. For each CS access, there are exactly $2(N-1)$ messages: $(N-1)$ REQUESTs and $(N-1)$ REPLYs.

```
• Initial local state for process $P_i$
    – int $My\_Sequence\_Number_i = 0$
    – int $ReplyCount_i = 0$
    – array of boolean $RD_i[j] = 0, \ \forall j \in \{1 \ldots N\}$
    – int $Highest\_Sequence\_Number\_Seen_i = 0$
• InvMutEx: Process $P_i$ executes the following to invoke mutual exclusion:
    1. $My\_Sequence\_Number_i = Highest\_Sequence\_Number\_Seen_i + 1$
    2. Make a REQUEST($R_i$) message, where $R_i = (My\_Sequence\_Number_i, \ i)$.
    3. Send this REQUEST message to all the other processes.
    4. $ReplyCount_i = 0$
    5. $RD_i[k] = 0, \ \forall k \in \{1 \ldots N\}$
• RcvReq: Process $P_i$ receives message REQUEST($R_j$), where $R_j = (SN, \ j)$, from process $P_j$:
    1. If $P_i$ is requesting then there are two cases.
        – $P_i$'s REQUEST has a higher priority than $P_j$'s REQUEST.
          In this case, $P_i$ sets $RD_i[j] = 1$ and
          $Highest\_Sequence\_Number\_Seen_i = max(Highest\_Sequence\_Number\_Seen_i, SN)$.
        – $P_i$'s REQUEST has a lower priority than $P_j$'s REQUEST. In this case, $P_i$ sends a REPLY
          to $P_j$.
    2. If $P_i$ is not requesting then send a REPLY message to $P_j$.
• RcvReply: Process $P_i$ receives REPLY message from process $P_j$:
    1. $ReplyCount_i = ReplyCount_i + 1$
    2. If (CheckExecuteCS) then execute CS.
• FinCS: Process $P_i$ finishes executing CS:
    1. Send REPLY to all processes $P_k$, such that $RD_i[k] = 1$.
• CheckExecuteCS: if ($ReplyCount_i = N - 1$) then return true else return false.
```

Fig. 1. Ricart-Agrawala algorithm.

3 Proposed Algorithm

3.1 Definitions

A REQUEST issued by process P_i with sequence number x is denoted using its ReqID as $R_{i,x}$. The priority of $R_{i,x}$ is the tuple (x, i), also denoted as $Pr(R_{i,x})$. The sequence number x is omitted whenever there is no ambiguity, and we say that a REQUEST R_i has a priority $Pr(R_i)$. This notation is used throughout this paper.

Two REQUESTs are said to be concurrent if for each requesting process, the REQUEST issued by the other process is received after the REQUEST has been issued by this process.

Definition 1. *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 P_j's REQUEST is received by P_i after P_i has made its REQUEST.*

Each REQUEST R_i sent by P_i has a concurrency set, denoted $CSet_i$, which is the set of those REQUESTs R_j that are concurrent with R_i. $CSet_i$ also includes R_i.

Definition 2. *Given R_i, $CSet_i = \{ R_j \mid R_i$ is concurrent with $R_j\} \cup \{R_i\}$.*

Observe that the relation "is concurrent with" is defined to be symmetric.

Observation 1. $R_i \in CSet_k$ iff $R_k \in CSet_i$.

3.2 Description and Basic Idea

The algorithm assumes the same model as the RA model. It also assumes that the underlying network channels are FIFO. The algorithm reduces the number of messages required per CS as compared to the RA algorithm. A process keeps a queue containing REQUESTs in the order of priorities, received by the process after it made its latest REQUEST. This queue, referred to as *Local_Request_Queue* (LRQ) (explained in Section 3.3), contains only concurrent REQUESTs. The algorithm uses three types of messages: REQUEST, REPLY, and FLUSH, and obtains savings by cleverly assigning multiple purposes to each. Specifically, these savings are obtained by the following key observations.

- All requests are totally ordered by priority, similar to the RA algorithm. A process receiving a REQUEST message can immediately determine whether the requesting process or itself should be allowed to enter the CS first.
- **Multiple uses of REPLY messages.**

  1. A REPLY message acts as reply from a process that is not requesting.
  2. A REPLY message acts as a collective reply from processes that have higher priority requests.

  A REPLY(R_j) message from P_j indicates that R_j is the REQUEST that P_j had last made and for which it executed the CS. This indicates that all REQUESTs which have priority $\geq$ the priority 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 queue. *Thus, a REPLY message is a logical reply and denotes a collective reply from all processes that had made higher priority requests.*
- **Uses of FLUSH message.** A FLUSH message is sent by a process after executing CS, to the **concurrently** requesting process with the next highest priority (if it exists). At the time of entering CS, a process can determine the state of all other processes in some possible consistent state with itself. Any other process is either requesting CS access and its requesting priority is known, or it is not requesting. At the time of finishing CS execution, a process P_i knows the following:

  1. Processes with concurrent lower (than P_i's) priority requests in P_i's local queue are waiting to execute CS.
  2. Processes which had sent REPLY to P_i for R_i are still not requesting, or are requesting with lower (than P_i's) priority.
  3. Processes which had requested concurrently with R_i with higher priority are not requesting or are requesting with a lower (than P_i's) priority.

  The REQUESTs received from processes identified in 2 and 3 are not concurrent with R_i, the REQUEST for which P_i just finished executing CS. Such REQUESTS received by P_i before it finishes CS are deferred until P_i finishes its CS. P_i then sends a REPLY to each of these deferred REQUESTs as soon as it finishes its CS.

  Thus, 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. For each process P_k identified in 2 and 3 that is requesting, its REQUEST would have been deferred until P_i left the CS, at which time P_i sends P_k a REPLY. With this behavior, P_i has given permission to both P_j and P_k that it is safe to enter CS with respect to P_i. P_j and P_k will have to get permission from one another, and the one with higher priority will enter the CS first.

  Similar to the R_i parameter on a REPLY message, the R_i parameter on the FLUSH denotes the ReqID, i.e., priority, of the REQUEST for which P_i just executed CS. When a process P_j receives FLUSH(R_i), it can remove those REQUESTs whose priority $\geq$ priority of R_i from its local queue. *Thus, a FLUSH message is a logical reply and denotes a collective reply from all processes that had made higher priority requests.*
- **Multiple uses of REQUEST messages.** A process P_i attempting to invoke mutual exclusion sends a REQUEST message to all other processes. Upon 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 as 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, then P_j's REQUEST, which reaches P_i after P_i has made its own REQUEST serves as a reply to P_i's REQUEST. After P_i executes the CS, P_j will receive a logical reply to its REQUEST through a chain of FLUSH/REPLY messages.

  Thus, in the proposed algorithm, concurrent REQUEST messages do not serve just the purpose of requesting. They are also some form of REPLY messages. The REQUEST message sent by P_i acts like an explicit reply to P_j's REQUEST if P_i's REQUEST has a lower priority than P_j's REQUEST.

  In the proposed algorithm as outlined above, a REQUEST message has three purposes, as summarized below. Assume that both P_i and P_j are requesting concurrently. Moreover, assume that the REQUEST of P_i has a higher priority than the REQUEST of P_j.

  1. A REQUEST message serves as a request message.
  2. The REQUEST message from P_i to P_j: This REQUEST message to P_j indicates to P_j that P_i is also in contention and has a higher priority. In this case, P_j should await FLUSH/REPLY from some process.
  3. The REQUEST message from P_j to P_i: This REQUEST message to P_i serves as a reply to P_i.

  Thus, no REPLY is sent when the REQUESTs are concurrent.

In the proposed algorithm, a process P_i requesting CS access by sending a REQUEST to other processes gets permission from any other process P_j, in one of the following ways:

- P_j is not requesting: P_j sends REPLY to P_i.
- P_j is concurrently requesting with a lower priority: P_j's REQUEST serves as the reply from P_j.
- P_j is concurrently requesting with a higher priority: P_j's REQUEST indicates that P_j is also in contention with a higher priority and that P_i should await FLUSH/REPLY, which transitively gives permission

to P_i. A FLUSH(R_k) or a REPLY(R_k) message, where $Pr(R_i) < Pr(R_k) \leq Pr(R_j)$, serves as permission from P_j to P_i.

3.3 The Algorithm

3.3.1 Data Structures for Process P_i

Each process P_i maintains the following data structures in addition to the local integer variables $My\_Sequence\_Number_i$ and $Highest\_Sequence\_Number\_Seen_i$.

- $RV_i[1:N]$ of Boolean. $RV_i[j] = 1$ indicates that process P_j has replied (by a REPLY or by a REQUEST or by a FLUSH). $RV_i[j] = 0$ indicates that process P_i has not yet replied.
- LRQ_i: queue of ReqIDs. This is a $Local\_Request\_Queue$ for ordering its own REQUEST and the concurrent requests (of lower and higher priority) from other processes that are received after P_i has made its own REQUEST.

A REPLY message from P_j also carries the ReqID of the last REQUEST made by P_j that was satisfied. Similarly, a FLUSH message from P_j carries the ReqID of the REQUEST for which P_j executed the CS. $Highest\_Sequence\_Number\_Seen$ is updated in a way similar to the RA algorithm.

3.3.2 Algorithm

The proposed algorithm is outlined in Fig. 2. Each procedure in the algorithm is executed atomically.

3.4 Example and Illustration of the Algorithm

3.4.1 An Example to Compare with RA

Fig. 3 shows an execution of processes P_1, P_2, and P_3 using a timing diagram from the time they attempt to enter the CS until all of them successfully execute CS. Assume that at time t_1 the highest sequence number at each process is zero. The status of LRQ and RV vectors at various instants of time shown in Fig. 3 are given below.

- Time instant t_1: None of the processes have sent out the REQUESTS:
  1. $RV_1 = [0,0,0]$ and $LRQ_1 = \langle\rangle$
  2. $RV_2 = [0,0,0]$ and $LRQ_2 = \langle\rangle$
  3. $RV_3 = [0,0,0]$ and $LRQ_3 = \langle\rangle$.
- Time instant t_2: All the processes have sent out the REQUESTS, but have not received a REQUEST/REPLY from any other process. The sequence number of these REQUESTs are one.
  1. $RV_1 = [1,0,0]$ and $LRQ_1 = \langle(1,1)\rangle$
  2. $RV_2 = [0,1,0]$ and $LRQ_2 = \langle(1,2)\rangle$
  3. $RV_3 = [0,0,1]$ and $LRQ_3 = \langle(1,3)\rangle$.
- Time instant t_3: All the processes have received REQUESTs from other processes.
  1. $RV_1 = [1,1,1]$ and $LRQ_1 = \langle(1,1),(1,2),(1,3)\rangle$
  2. $RV_2 = [1,1,1]$ and $LRQ_2 = \langle(1,1),(1,2),(1,3)\rangle$
  3. $RV_3 = [1,1,1]$ and $LRQ_3 = \langle(1,1),(1,2),(1,3)\rangle$.

  Note that P_1 does not send any REPLY to P_2 and P_3. Instead, $CheckExecuteCS$ returns *true* and P_1 executes CS. Similarly P_2 also does not send any REPLY to P_1 and P_3. Moreover, $CheckExecuteCS$ returns *false*, and so P_2 cannot execute CS. Note the difference when compared to RA. As per RA, P_2 will send a REPLY to P_1. P_3 also does not send any REPLY to P_1 and P_2. $CheckExecuteCS$ returns *false*, and so P_3 cannot execute CS. Once again, note the difference when compared to RA. As per RA, P_3 will send a REPLY to P_1 and P_2.
- Time instant t_4: P_1 finishes CS. Other processes are waiting to execute CS.
  1. $RV_1 = [1,1,1]$ and $LRQ_1 = \langle(1,1),(1,2),(1,3)\rangle$
  2. $RV_2 = [1,1,1]$ and $LRQ_2 = \langle(1,1),(1,2),(1,3)\rangle$
  3. $RV_3 = [1,1,1]$ and $LRQ_3 = \langle(1,1),(1,2),(1,3)\rangle$.

  After P_1 finishes executing CS, it examines LRQ_1 and determines the next request in LRQ_1. It is P_2. P_1 sends a FLUSH((1,1)) message to P_2. Note that the parameter indicates the ReqID of the request for which P_1 executed CS. This action is different from RA. In RA, P_1 will send REPLY messages to P_2 and P_3.
- Time instant t_5: P_2 gets the FLUSH((1,1)) message.
  1. $RV_2 = [1,1,1]$ and $LRQ_2 = \langle(1,1),(1,2),(1,3)\rangle$
  2. $RV_3 = [1,1,1]$ and $LRQ_3 = \langle(1,1),(1,2),(1,3)\rangle$

  When P_2 gets the FLUSH((1,1)) message, it finds the entry (1,1) in LRQ_2. P_2 removes all entries ahead of (1,1) and including (1,1). Now *CheckExecuteCS* returns *true* and P_2 executes CS.
- Time instant t_6: P_2 finishes CS.
  1. $RV_2 = [1,1,1]$ and $LRQ_2 = \langle(1,2),(1,3)\rangle$
  2. $RV_3 = [1,1,1]$ and $LRQ_3 = \langle(1,1),(1,2),(1,3)\rangle$

  After P_2 finishes executing CS, it examines its LRQ and determines the next request in LRQ_2. It is P_3. P_2 sends a FLUSH((1,2)) message to P_3.

Actions of P_3, when it receives the FLUSH message, are similar to the actions of P_2. After P_3 finishes executing CS, it does not send any FLUSH/REPLY to any other process.

For the three requests in Fig. 3, the RA algorithm needs $3 \cdot (2 \cdot (N-1)) = 12$ messages to enforce mutual exclusion. In the proposed algorithm, only eight messages are required, thus saving 33 percent over the RA algorithm.

3.4.2 Some Example Scenarios

Fig. 4 gives an example scenario at process P_4 in a system consisting of six processes $P_1, P_2, \ldots, P_6$. Fig. 4a shows the scenario when P_4 has just sent its REQUESTs. P_4's REQUEST is the only REQUEST in LRQ_4 and RV_4 is $[0,0,0,1,0,0]$. Fig. 4b shows the scenario at some time after P_4 has requested and before P_4 executes CS. P_4 has received higher priority REQUESTs from P_1 and P_3 and a lower priority REQUEST from P_5. It has also received a REPLY from P_2. It has not received any message from P_6, either in

```
• Initial local state for process $P_i$
    – int $My\_Sequence\_Number_i = 0$
    – array of boolean $RV_i[j] = 0,\ \forall j \in \{1 \ldots N\}$
    – queue of ReqId $LRQ_i$ is NULL
    – int $Highest\_Sequence\_Number\_Seen_i = 0$
• InvMutEx: Process $P_i$ executes the following to invoke mutual exclusion:
    1. $My\_Sequence\_Number_i = Highest\_Sequence\_Number\_Seen_i + 1$
    2. $LRQ_i = NULL$.
    3. Make a REQUEST($R_i$) message, where $R_i = (My\_Sequence\_Number_i,\ i)$.
    4. Insert this REQUEST in the $LRQ_i$ in sorted order.
    5. Send this REQUEST message to all the other processes.
    6. $RV_i[k] = 0\ \forall k \in \{1 \ldots N\} - \{i\}$. $RV_i[i] = 1$.
• RcvReq: Process $P_i$ receives REQUEST($R_j$), where $R_j = (SN,\ j)$, from process $P_j$:
    1. $Highest\_Sequence\_Number\_Seen_i = max(Highest\_Sequence\_Number\_Seen_i, SN)$.
    2. If $P_i$ is requesting:
        (a) if $RV_i[j] = 0$, then insert this request in the $LRQ_i$ (in sorted order) and mark $RV_i[j] = 1$.
            If ($CheckExecuteCS$) then execute CS.
        (b) if $RV_i[j] = 1$, then defer the processing of this REQUEST, which will be processed after $P_i$
            executes CS.
    3. If $P_i$ is not requesting then send a REPLY($R_i$) message to $P_j$. $R_i$ denotes the ReqID of the last
       request made by $P_i$ that was satisfied.
• RcvReply: Process $P_i$ receives REPLY($R_j$) message from process $P_j$: $R_j$ denotes the ReqID of the last
  request made by $P_j$ that was satisfied.
    1. $RV_i[j] = 1$
    2. Remove all REQUESTs from $LRQ_i$ that have the priority $\geq$ the priority of $R_j$.
    3. If ($CheckExecuteCS$) then execute CS.
• FinCS: Process $P_i$ finishes executing CS.
    1. Send FLUSH($R_i$) message to the next candidate in the $LRQ_i$. $R_i$ denotes the ReqID that was
       satisfied.
    2. Send a REPLY($R_i$) to the deferred REQUESTs. $R_i$ is the ReqID corresponding to which $P_i$ just
       executed CS.
• RcvFlush: Process $P_i$ receives a FLUSH($R_j$) message from process $P_j$:
    1. $RV_i[j] = 1$
    2. Remove all requests in $LRQ_i$ that have the priority $\geq$ the priority of $R_j$.
    3. If ($CheckExecuteCS$) then execute CS.
• $CheckExecuteCS$: if ($RV_i[k] = 1,\ \forall k \in \{1 \ldots N\}$) and $P_i$'s request is at the head of $LRQ_i$ then return
  true else return false.
```

Fig. 2. The proposed algorithm.

the form of a REPLY or a REQUEST. Fig. 4c shows the scenario when the process P_4 is about to enter the CS. $CheckExecuteCS$ returns *true* in this scenario. The intermediate steps between Figs. 4a, 4b, and 4c are not shown.

Figs. 5, 6, and 7 illustrate the algorithm via some example scenarios using timing diagrams. Time lines of processes are shown horizontally. A vertical line intersecting the horizontal time line of a process indicates that the adjacent comment applies to the event at the intersection point.

Fig. 5 shows some possible scenarios in a system of three processes. Process identifiers are ordered as follows: $i < j < k$.

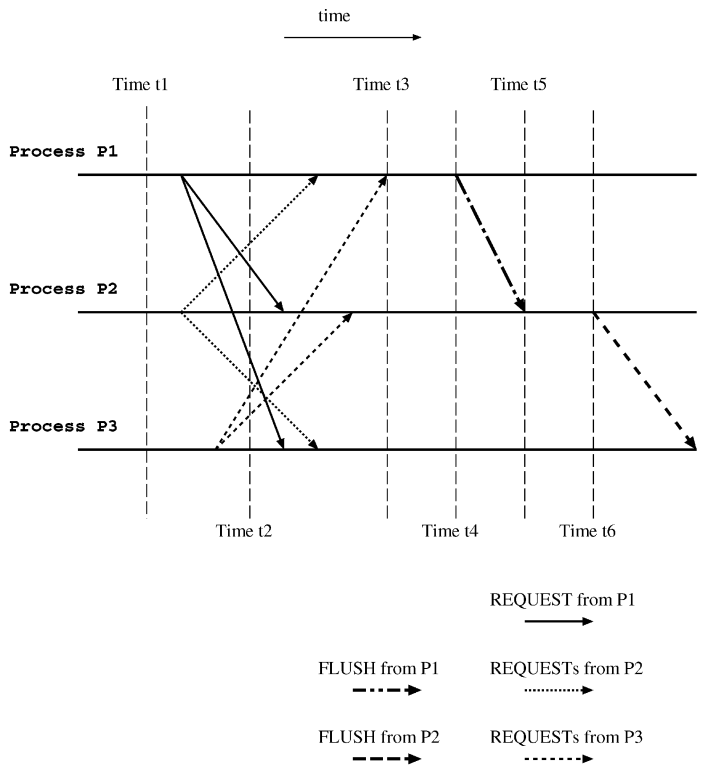


Fig. 3. An illustrative example of the algorithm used to compare with the RA algorithm.

- **Scenario 1.** P_i is the only process that is requesting CS access. P_i gets REPLY messages from P_j and P_k. After getting these REPLY messages, P_i can execute CS.
- **Scenario 2a.** P_i and P_j are requesting CS access concurrently. P_j's REQUEST has a lower priority than P_i's REQUEST. P_j's REQUEST acts as a reply to P_i. P_i sends a FLUSH to P_j after executing CS. This FLUSH acts as a logical reply to P_j for P_j's REQUEST. P_j executes CS on getting the FLUSH from P_i.
- **Scenario 2b.** P_i and P_j are requesting (not concurrently). P_i gets a REPLY from P_j. It then gets a REQUEST from P_j while it is waiting for a REPLY/REQUEST from P_k. As $RV_i[j] = 1$, this REQUEST of P_j is deferred and later processed after P_i finishes executing CS. When P_i processes this REQUEST, it sends a REPLY to P_j.
- **Scenario 2c.** P_i and P_j are requesting CS access concurrently. P_i executes CS and sends a FLUSH message to P_j. P_j is still awaiting a REPLY/REQUEST from P_k. So P_j cannot execute CS on getting this FLUSH. While it is waiting for a REPLY/REQUEST from P_k, it gets another REQUEST from P_i. Since $RV_j[i]$ is 1, it defers this REQUEST. On getting a REPLY from P_k, P_j executes CS. After P_j finishes executing CS, it processes the deferred REQUEST by sending a REPLY to it.

Fig. 6 illustrates a scenario where a REPLY message acts as a logical reply from all higher priority requesting processes. P_i, P_j, and P_k are the processes in the system that are requesting CS access. Priority of P_i's REQUEST is the highest and priority of P_k's REQUEST is the lowest. P_i's REQUEST is concurrent with the requests from P_k and P_j. However, P_j's REQUEST is made causally before P_k's REQUEST. P_i's REQUEST is just ahead of P_k's REQUEST in LRQ_k. Observe that P_k will not get a FLUSH message from P_i because after P_i executes CS, P_j's REQUEST is just behind P_i's REQUEST in LRQ_i and therefore P_i sends a FLUSH to P_j and not to P_k. As per the algorithm, P_k will also not receive a REPLY message from P_i. P_j executes CS on receiving a FLUSH from P_i. P_j sends a REPLY to the deferred request from P_k after P_j finishes executing CS. P_k gets this REPLY(R_j) from P_j, where R_j denotes the REQUEST of P_j that was last satisfied. P_k deletes all entries in LRQ_k that have a priority $\geq$ the priority of R_j (algorithm step *RcvReply*.2). This deletes P_i's REQUEST from LRQ_k and makes P_k's REQUEST the head of LRQ_k. P_k can now execute CS.

Fig. 7 illustrates a scenario where process P_k receives more than one FLUSH message. The first REQUEST of P_i is not concurrent with the REQUEST from P_k and the second REQUEST from P_i has a higher priority than P_k's REQUEST. The order of CS executions is as follows: P_i executes CS first, then P_j, then P_i again, and then P_k. P_k receives two FLUSH messages, the first from P_j and the second from P_i before it can execute CS. P_j sends P_k a FLUSH because when P_j finishes its CS, P_k's REQUEST is just behind P_j's REQUEST in LRQ_j and P_j has not yet received P_i's second REQUEST. Even after receiving the FLUSH from P_j, P_k cannot execute CS unless it receives permission from P_i also. As P_i's REQUEST has greater

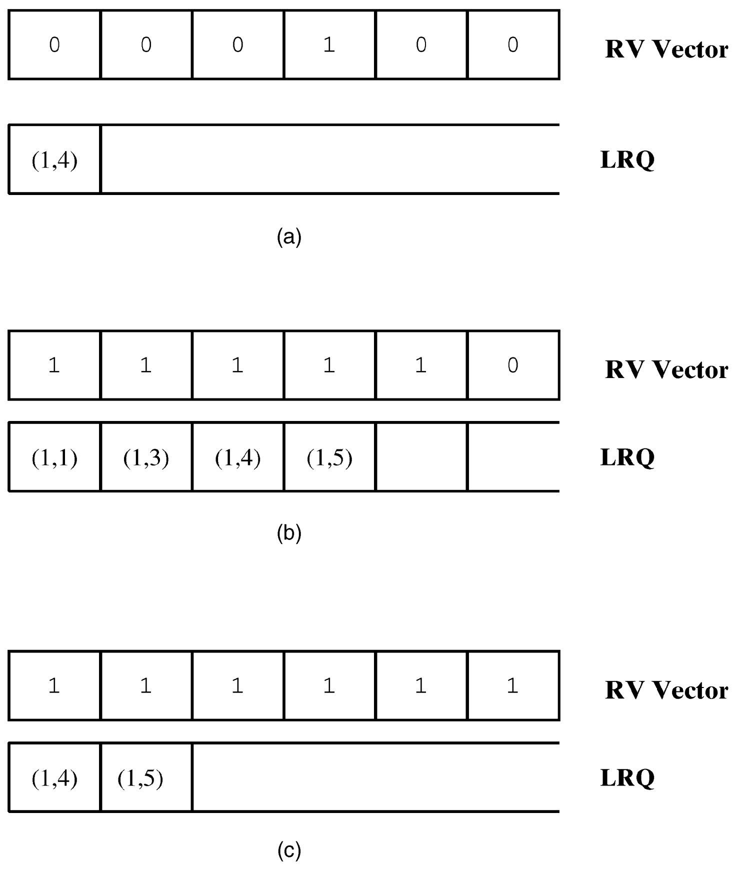


Fig. 4. An example scenario at process P4. (a) RV Vector and LRQ at process P4, just after P4 sends its REQUEST. (b) RV Vector and LRQ at process P4, some time after P4 sends its REQUEST. (c) RV Vector and LRQ at process P4, just before entering CS.

priority than P_k's REQUEST, P_k has to await a direct or a logical FLUSH/REPLY from P_i. In the given scenario, P_i sends a FLUSH to P_k after executing CS because P_k's lower priority REQUEST was received before P_i entered its CS. Only after getting the FLUSH from P_i can P_k execute CS.

4 Analysis and Correctness Proof

4.1 Message Complexity

The number of messages per CS access can be deterministically expressed as a measure of concurrency of REQUESTs as follows. A site P_i that is requesting sends $(N-1)$ REQUEST messages. It receives $(N- \mid CSet_i \mid)$ REPLYs. There are two cases to consider.

1. $\mid CSet_i \mid \geq 2$. There are two subcases here.
   a. There is at least one REQUEST in $CSet_i$ whose priority is less than that of R_i. So P_i will send one FLUSH message. In this case, the total number of messages for CS access is $2N- \mid CSet_i \mid$. When all REQUESTs are concurrent, this is N messages.
   b. There is no REQUEST in $CSet_i$, whose priority is less than the priority of R_i. P_i will not send a FLUSH message. In this case, the total number of messages for CS access is $2N-1- \mid CSet_i \mid$. When all REQUESTs are concurrent, this is $N-1$ messages.
2. $\mid CSet_i \mid = 1$. This is the worst case, implying that all REQUESTs are serialized. P_i will not send a FLUSH message. In this case, the total number of messages for CS access is $2(N-1)$ messages.

4.2 Definitions Used in the Proof

We give some definitions and then an observation on a property of the algorithm. These definitions and the observation are used to prove the correctness of the algorithm.

Definition 3 defines the concept of a predecessor of a REQUEST R_i in a set S of REQUESTs.

Definition 3.

$$\begin{aligned} Pred(R_i, S) =& R_j \ iff \ R_j \in S \ \wedge \ Pr(R_i) < Pr(R_j) \\ & \wedge \ \nexists R_k \in S \mid (Pr(R_i) < Pr(R_k) < Pr(R_j)). \end{aligned}$$

Definition 4 defines the concept of a successor of a REQUEST R_i in a set S of REQUESTs.

Definition 4.

$$\begin{aligned} Succ(R_i, S) =& R_j \ iff \ R_j \in S \ \wedge \ Pr(R_i) > Pr(R_j) \\ & \wedge \ \nexists R_k \in S \mid (Pr(R_j) < Pr(R_k) < Pr(R_i)). \end{aligned}$$

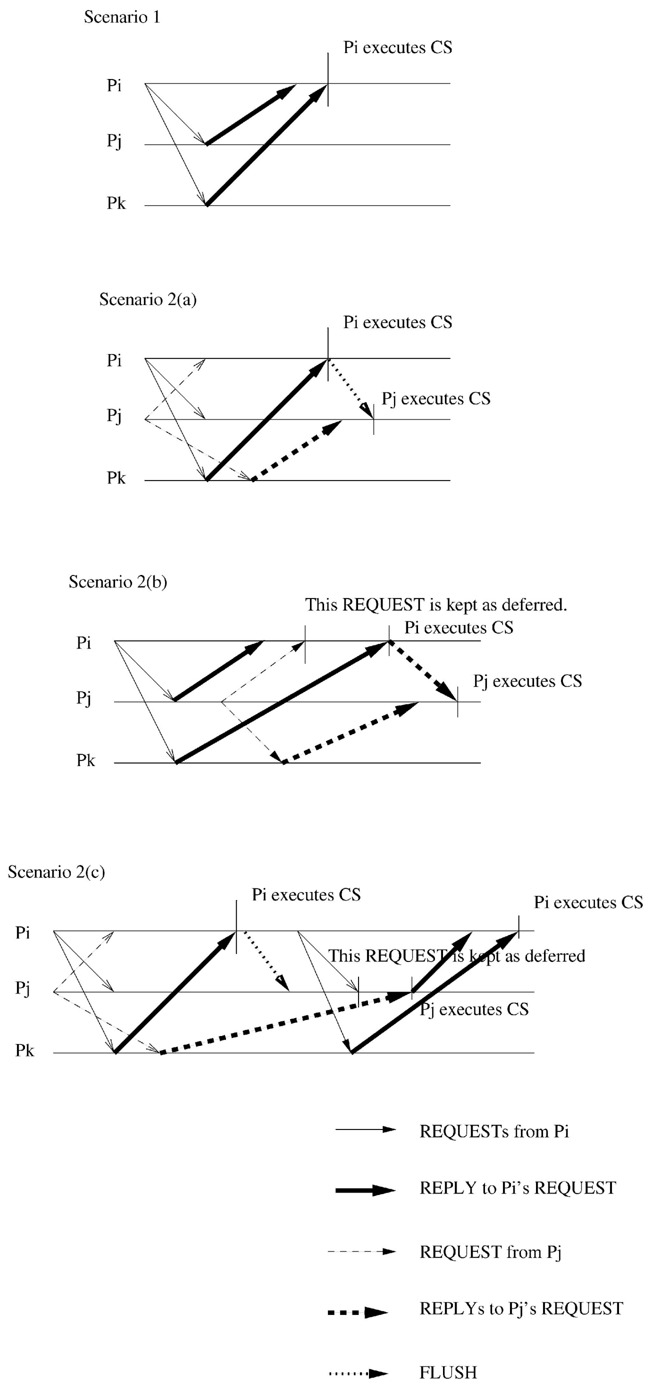


Fig. 5. Some illustrative scenarios using timing diagrams.

Definition 5 defines a global view (GV) of the system execution. GV_{R_i,R_j} is the set of REQUESTs R_k ever made in the system execution, whose priority lies in the range $[Pr(R_i), Pr(R_j)]$. Although the global view of REQUESTs may not be available to any process, nonetheless it can be assumed to be available for the purpose of proving the correctness of the algorithm.

Definition 5. $GV_{R_i,R_j} = \{R_k \mid Pr(R_j) \leq Pr(R_k) \leq Pr(R_i)\}$.

Definition 6 defines a notion of distance ($Dist$) between two REQUESTs $R_i, R_j \in GV_{R_1,R_2}$. $Dist(R_i, R_j)$ is defined as $1+$ the number of REQUESTs that have a priority value greater than $Pr(R_j)$ and less than $Pr(R_i)$.

Definition 6. $Dist(R_i, R_j) = |GV_{R_i,R_j}| - 1$.

Given two REQUESTs R_i and R_k such that each is in the concurrency set of the other requesting process (Observation 1) and that they are at a distance of one in the global view $(Pr(R_i) > Pr(R_k))$, then R_i is the predecessor of R_k in P_k's concurrency set and R_k is the successor of R_i in P_i's concurrency set. This is captured by Observation 2.

Observation 2. The two parts of this observation are as follows.

1.

$$Dist(R_i, R_k) = 1 \wedge R_k \in CSet_i \Longrightarrow (Pred(R_k, CSet_k) = R_i \wedge\ Succ(R_i, CSet_i) = R_k).$$

2.

$$Dist(R_i, R_k) = 1 \ \wedge\ R_i \in CSet_k \Longrightarrow (Pred(R_k, CSet_k) = R_i \wedge\ Succ(R_i, CSet_i) = R_k).$$

4.3 Safety and Fairness

A mutual exclusion algorithm satisfies the safety specification of the mutual exclusion problem if it provides mutually exclusive access to the critical section. A (safe) mutual exclusion algorithm is said to provide fair mutual exclusion if the following property holds.

Definition 7. *An algorithm provides fair mutual exclusion iff $Pr(R_i) > Pr(R_j) \Longleftrightarrow P_j$ executes CS after P_i finishes CS.*

Theorem 1 (Safety and Fairness). *The algorithm in Fig. 2 provides fair mutual exclusion as defined in Definition 7.*

Proof. Let R_a be the REQUEST that has the highest priority among all REQUESTs ever made and R_b be the REQUEST that has the lowest priority among all REQUESTs ever made until now. We will prove that for any two REQUESTs R_i, $R_j \in GV_{R_a,R_b}$ such that $Pr(R_j) > Pr(R_i)$, P_i enters CS after P_j finishes CS.

The proof is by induction on $Dist(R_a, R_i)$, i.e., for any R_i such that $Dist(R_a, R_i) > 0$, P_i executes CS next after P_j finishes CS, where $R_j = Pred(R_i, GV_{R_a,R_b})$.

Induction hypothesis. For any REQUEST $R_i \in GV_{R_a,R_b}$ such that $Dist(R_a, R_i) > 0$, P_i executes CS next after P_j finishes CS, where $Pred(R_i, GV_{R_a,R_b}) = R_j$.

Base case $Dist(R_a, R_i) = 1$. We need to prove that P_i executes CS next after P_a finishes CS. There are two cases here.

- $R_i \in CSet_a$. By Observations 1 and 2,

$$Succ(R_a, CSet_a) = R_i$$

and $Pred(R_i, CSet_i) = R_a$. P_a will send a FLUSH to P_i on finishing CS (algorithm step $FinCS.1$). There is no other REQUEST R_k in the system such

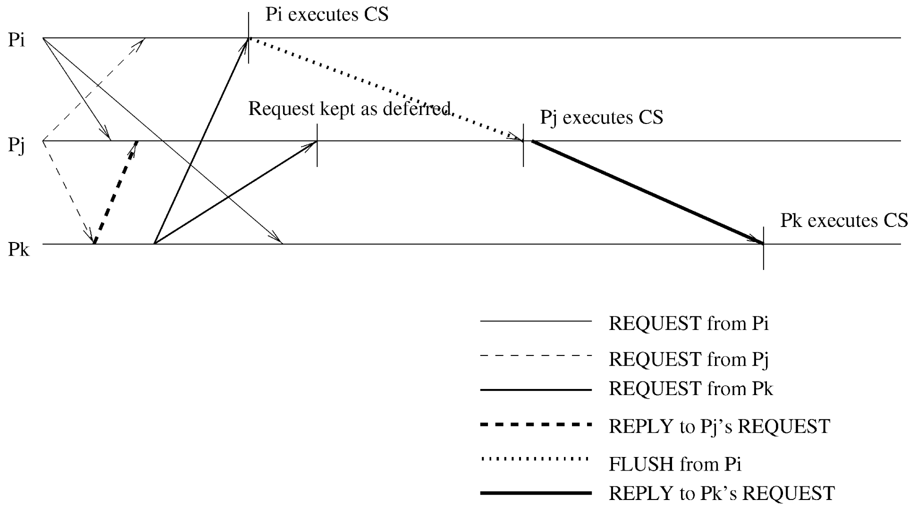


Fig. 6. A timing diagram for a scenario where REPLY acts as a logical reply from multiple processes.

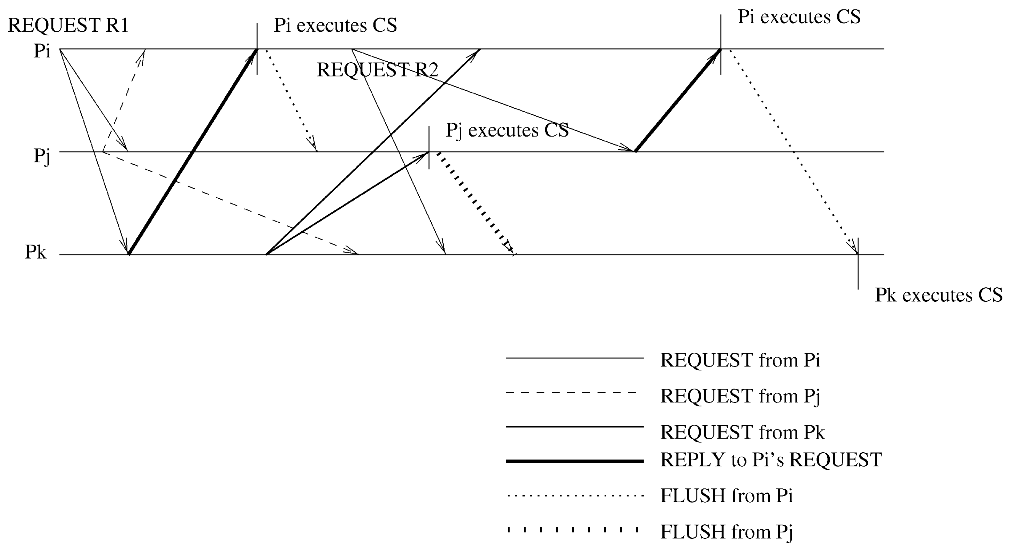


Fig. 7. A timing diagram for a scenario where a process receives more than one FLUSH.

that $Pr(R_i) < Pr(R_k) < Pr(R_a)$. A FLUSH comes from a higher priority process only. P_i cannot execute CS unless P_i gets a FLUSH from P_a. When P_i receives FLUSH from P_a (algorithm step *RcvFlush*), it deletes R_a from LRQ_i and can execute CS if it has received logical replies from other processes. Any other process is either requesting with a lower priority, in which case its REQUEST serves as a reply, or is not requesting, in which case it sends a REPLY.

We now need to prove that no REPLY(R_k) is received by P_i, such that $Pr(R_i) < Pr(R_k) < Pr(R_a)$. (Recall that R_k is the ReqID of the last REQUEST, corresponding to which the process P_k executed CS). This follows from the fact that such an R_k does not exist in the global view. Moreover, we also need to prove that no REPLY(R_k) is received by P_i in response to R_i such that $Pr(R_k) < Pr(R_i)$. There are three subcases here.

1. If R_k was received by P_i, before P_i had sent R_i, then $Pr(R_i) < Pr(R_k)$. This is a contradiction.
2. R_i and R_k are concurrent. There are two subcases here.

   a. R_i has lower priority, which is a contradiction.
   b. R_i has a higher priority. Then R_k acts as a reply to P_i's REQUEST. This is a contradiction, as P_k sends an explicit REPLY to P_i.
3. R_k is issued after P_k receives R_i. This is a contradiction, because P_k cannot send a REPLY(R_k), as P_k has not yet executed CS corresponding to the REQUEST R_k.

Thus, no REPLY (R_k) is received by P_i in response to R_i such that $Pr(R_k) < Pr(R_i)$. So P_i executes CS next after P_a finishes CS.

- $R_i \notin CSet_a$. Requests R_i and R_a are not concurrent. As R_i has a lower priority than R_a, P_i requested only after receiving a REQUEST from P_a and sending back a REPLY to P_a. There are two subcases here.
    - P_a receives R_i before entering CS. In this case, P_a defers the REQUEST R_i and processes it only after finishing CS because it has already received a REPLY from P_i (algorithm step *RcvReq2b*).
    - P_a receives R_i after finishing CS. (A request received while executing CS is deferred until CS completion.)

  In both subcases, P_i can execute CS only after P_a finishes CS (detected by P_i when it receives REPLY from P_a) and if P_i has received logical replies from other processes. Any other process is either requesting with a lower priority, in which case its REQUEST serves as a reply, or is not requesting, in which case it sends a REPLY. Analogous to the proof for the case of $R_i \in CSet_a$, P_i will not receive any REPLY(R_k) in response to R_i such that either $Pr(R_i) < Pr(R_k) < Pr(R_a)$ or $Pr(R_k) < Pr(R_i)$. Thus, P_i executes CS next after P_a finishes CS.

Induction step $Dist(R_a, R_i) = x,\ x > 1$. For a REQUEST $R_i \in GV_{R_a,R_b}$ such that $Dist(R_a, R_i) = x$, P_i executes CS next after P_j finishes CS, where $Pred(R_i, GV_{R_a,R_b}) = R_j$.

Induction step $Dist(R_a, R_i) = x + 1,\ x > 1$. For a REQUEST $R_i \in GV_{R_a,R_b}$ such that $Dist(R_a, R_i) = x + 1$, P_i executes CS next after P_j finishes CS, where $Pred(R_i, GV_{R_a,R_b}) = R_j$. By the induction hypothesis, we claim that P_j executes CS next after P_k finishes CS, where $Pred(R_j, GV_{R_a,R_b}) = R_k$.

To complete this proof, we need to prove that P_i executes CS next after P_j finishes CS.

- $R_i \in CSet_j$. This case is similar to the corresponding base case where $Dist(R_a, R_i) = 1$.
- $R_i \notin CSet_j$. Analogous to the corresponding base case where $Dist(R_a, R_i) = 1$, P_j sends a REPLY to P_i's REQUEST only after finishing CS. By the induction hypothesis, all requests with priority greater than $Pr(R_i)$ have been served. Any other process P_m is either requesting with a lower priority than $Pr(P_i)$, in which case its REQUEST serves as a reply, or is not requesting, in which case it sends a REPLY(R_m), where neither $Pr(R_i) < Pr(R_m) < Pr(R_j)$ nor $Pr(R_m) < Pr(R_i)$. So P_i executes CS after P_j finishes CS and P_i has received a logical reply from all other processes.

In both cases, P_i executes CS next after P_j finishes CS.

1) We showed the proof using induction on $Dist(R_a, R_i)$. In the global view, all REQUESTs are totally ordered. Hence, at any distance $Dist(R_a, R_i)$, there is a unique R_i. As every REQUEST in the system has a unique priority, it is at a unique distance $Dist(R_a, R_i)$. REQUESTs are satisfied in the order of increasing distance (decreasing priority). Hence, if $Pr(R_i) > Pr(R_j)$, then P_j executes after P_i finishes CS.

2) From 1) and the fact that each REQUEST in the system has a unique priority, we can say that if P_j executes CS after P_i finishes CS, then $Pr(R_i) > Pr(R_j)$.

From 1) and 2), fair mutual exclusion is guaranteed by the algorithm. □

4.4 Liveness

Liveness is achieved if any process that requested CS access executes CS eventually.

Theorem 2 (Liveness). *The algorithm in Fig. 2 achieves liveness.*

Proof. Let R_a be the REQUEST that has the highest priority among all REQUESTs ever made. Let R_b be the REQUEST that has the lowest priority among all REQUESTs ever made until now. We first prove that P_a executes CS. We then prove by induction that for any REQUEST R_k, such that $Dist(R_a, R_k) \geq 1$, process P_k executes CS.

As R_a is the highest priority REQUEST in the system, P_a must have received either a low priority concurrent REQUEST or a REPLY from each other process. It will not receive any higher priority REQUESTs. Moreover, it will not get any FLUSH, which can arrive only from a higher priority process. So P_a executes CS. We prove by induction that for any $Dist(R_a, R_k) > 0, P_k$ executes CS.

Induction hypothesis. For any $Dist(R_a, R_k) > 0, P_k$ executes CS.

Base case $Dist(R_a, R_k) = 1$. If $Dist(R_a, R_k) = 1$, then P_k executes CS. There are two cases here.

- $R_k \in CSet_a$. By Observations 1 and 2, $Succ(R_a, CSet_a) = R_k$ and $Pred(R_k, CSet_k) = R_a$. We have shown that P_a executes CS. After executing CS, it sends a FLUSH to P_k (algorithm step *FinCS*). As $Pred(R_k, CSet_k) = R_a$, on getting a FLUSH from P_a (algorithm step *RcvFlush*), P_k is at the head of LRQ_k and can execute CS if it receives replies from other processes in the form of REQUESTs, REPLYs, or FLUSHs. Any other process is either requesting with a lower priority, in which case its REQUEST serves as a reply, or is not requesting, in which case it sends a REPLY. Thus, P_k executes CS.
- $R_k \notin CSet_a$. The REQUESTs R_k and R_a are not concurrent. So P_k requested only after receiving a REQUEST from P_a and returning a REPLY, implying that R_k has lower priority that R_a. There are two subcases here:
    - P_a receives R_k before entering CS. In this case, P_a defers the REQUEST R_k and processes it only after finishing CS. After finishing CS, it sends a REPLY to P_k. This REPLY enables P_k to execute CS if it has received

logical replies from all other processes (algorithm step *RcvReply*).

- P_a receives R_k after finishing CS. (A request received during CS execution is deferred until CS completion.) P_a sends a REPLY to P_k. This REPLY enables P_k to execute CS if it has received logical replies from all other processes (algorithm step *RcvReply*).

In both subcases, any other process is either requesting with a lower priority, in which case its REQUEST serves as a reply or is not requesting, in which case it sends a REPLY. Thus, P_k will execute CS.

Induction step $Dist(R_a, R_k) = x,\ x > 1$. We assume that P_k executes CS.

Induction step $Dist(R_a, R_k) = x + 1,\ x > 1$. Let R_m be such that $Dist(R_a, R_m) = x$. Then $Dist(R_m, R_k) = 1$. From the induction hypothesis, we claim that P_m executes CS. We need to prove that P_k executes CS if R_m executes CS. Similar to the base case, there are two cases here:

- $R_k \in CSet_m$. This case is similar to the corresponding base case where $Dist(R_a, R_k) = 1$. Thus, P_k executes CS.
- $R_k \notin CSet_m$. Analogous to the corresponding base case, where $Dist(R_a, R_k) = 1$, P_k will get a REPLY from P_m. Moreover, there is no REQUEST that has a priority in between the priority of R_m and R_k. When R_k gets this REPLY(R_m), it will remove all REQUESTs from its LRQ_k that have priority higher than or equal to Pr(R_m) (algorithm step *RcvReply*). This will make R_k the head of LRQ_k. Any other process is either requesting with a lower priority, in which case its REQUEST serves as a reply, or is not requesting, in which case it sends a REPLY. Thus, P_k can execute CS.

Thus, P_k executes CS and the algorithm guarantees liveness. $\square$

5 Discussion and Concluding Remarks

We presented a fair mutual exclusion algorithm for a distributed system with asynchronous message passing. Fairness is defined in terms of satisfying requests for CS access in decreasing order of priority, which is defined by Lamport's timestamp. This algorithm requires between $[N-1, 2(N-1)]$ messages per access to the critical section, and improves upon the Ricart-Agrawala algorithm, which is the best known fair algorithm, without introducing any additional overhead. Specifically, the number of messages for a CS access is $2(N-1) - x$, where x is the number of other requests that are made concurrently with this request. The savings in message complexity was obtained by exploiting the concurrency of requests and assigning multiple meanings to the requests and replies whenever there are concurrent requests. The algorithm as presented here is not resilient to node or link failures. However, this is also a drawback of Lamport's algorithm and the RA algorithm.

The following improvements can be made to the algorithm. The first improvement saves on the number of REPLY messages sent. Observe that a process P_i on finishing CS (procedure $FinCS$) sends a FLUSH to the concurrently requesting process with the next highest priority (if it exists) and REPLYs (say m) to the processes whose REQUESTs were deferred. By examining these REQUESTs, P_i can determine the relative order in which these processes will execute CS. Using this fact, the following optimization can be made. Assume P_k has the highest priority among these REQUESTs. P_i can send REPLY just to P_k, apprising P_k of all the information P_i has gathered. Thus P_i can avoid sending upto m (worst case is $m-1$) messages. Now it is upto P_k to take care of the rest. However, this optimization requires a significant increase in message sizes and local data structures.

A second way to save on the number of REPLY messages is by treating deferred REQUESTs as concurrent to the next REQUEST of this process (although they are not truly concurrent by definition). If the process exiting the CS knows that it will be requesting CS access soon, it can keep the deferred REQUESTs as deferred until it makes its next REQUEST. At that time, its REQUEST acts as a REPLY to the deferred REQUESTs, and the deferred REQUESTs act a REPLY to its REQUEST. This optimization could slow down the computation at processes.

A third improvement is as follows: The $Highest\_Sequence\_Number\_Seen$ behaves as a global function of the sequence number of requests and is used as a determinant of the priority of each request for CS access. The fair algorithm satisfies requests in order of decreasing priority. In the presented algorithm,

$$Highest\_Sequence\_Number\_Seen$$

is a parameter only on REQUEST messages, akin to the Lamport and the Ricart-Agrawala algorithm. In order that the priority be determined most fairly, taking into account the transitive causality relation among events induced by all messages exchanged, the $Highest\_Sequence\_Number\_Seen$ can be introduced as a parameter on all algorithm messages.

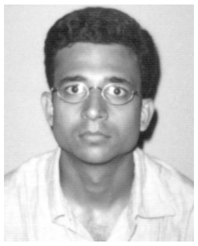
Acknowledgments

Ajay Kshemkalyani's work was supported by U.S. National Science Foundation grant CCR-9875617.

References

[1] O. Carvalho and G. Roucairol, "On Mutual Exclusion in Computer Networks, Technical Correspondence," *Comm. ACM*, vol. 26, no. 2, pp. 146-147, Feb. 1983.

[2] Y.-I. Chang, "A Simulation Study on Distributed Mutual Exclusion," *J. Parallel and Distributed Computing,* vol. 33, pp. 107-121, 1996.
[3] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[4] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Trans. Computer Systems,* vol. 3, no. 2, pp. 145-159, May 1985.
[5] M. Naimi, M. Trehel, and A. Arnold, "A $\log(n)$ Distributed Mutual Exclusion Algorithm Based on Path Reversal," *J. Parallel and Distributed Computing,* vol. 34, pp. 1-13, 1996.
[6] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Trans. Computer Systems,* vol. 7, no. 1, pp. 61-77, Feb. 1989.
[7] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM,* vol. 24, no. 1, pp. 9-17, Jan. 1981.
[8] B. Sanders, "The Information Structure of Distributed Mutual Exclusion Algorithms," *ACM Trans. Computer Systems,* vol. 5, no. 3, pp. 284-299, Aug. 1987.
[9] M. Singhal, "A Taxonomy of Distributed Mutual Exclusion," *J. Parallel and Distributed Computing,* vol. 18, no. 1, pp. 94-101, May 1993.
[10] M. Singhal, "A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 1, pp. 121-125, Jan. 1992.
[11] M. Singhal, "A Heuristically Aided Algorithm for Mutual Exclusion in Distributed Systems," *IEEE Trans. Computers,* vol. 38, no. 5, pp. 651-662, May 1989.
[12] I. Suzuki and T. Kasami, "A Distributed Mutual Exclusion Algorithm," *ACM Trans. Computer Systems,* vol. 3, no. 4, pp. 344-349, Nov. 1985.
[13] Y. Yan, X. Zhang, and H. Yang, "A Fast Token-Chasing Mutual Exclusion Algorithm in Arbitrary Network Topologies," *J. Parallel and Distributed Computing,* vol. 35, pp. 156-172, 1996.
[14] J.-H. Yang and J. Anderson, "Time Bounds for Mutual Exclusion and Related Problems," *Proc. 26th Ann. ACM Symp. Theory of Computing,* pp. 224-233, May 1994.
[15] J.-H. Yang, J. Anderson, "A Fast, Scalable Mutual Exclusion Algorithm," *Distributed Computing* vol. 9, no. 1, pp. 51-60, Aug. 1995.



Sandeep Lodha received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, India in 1997, and the MS degree in computer engineering from the University of Cincinnati in 2000. Currently, he is working at Synopsys, Inc., in Mountain View, California. His research interests include parallel and distributed systems, physical design automation of VLSI systems, high level synthesis, VLSI design, and CAD for reconfigurable systems.

Ajay Kshemkalyani received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the PhD degree in computer and information science from the Ohio State University in 1991. From Fall 2000, he is an associate professor of electrical engineering and computer science at the University of Illinois at Chicago. He has also worked at IBM, Research Triangle Park, North Carolina, in computer networks and distributed systems. His current research interests are in distributed computing, computer networking, and operating systems. He is a member of the ACM and a senior member of the IEEE.