A Delay-Optimal Quorum-Based Mutual Exclusion Algorithm for Distributed Systems

Guohong Cao, *Member*, *IEEE*, and Mukesh Singhal, *Fellow*, *IEEE*

Abstract—The performance of a mutual exclusion algorithm is measured by the number of messages exchanged per critical section execution and the delay between successive executions of the critical section. There is a message complexity and synchronization delay trade-off in mutual exclusion algorithms. The Lamport algorithm and the Ricart-Agrawal algorithm both have a synchronization delay of T (T is the average message delay), but their message complexity is $O(N)$. Maekawa's algorithm reduces the message complexity to $O(\sqrt{N})$; however, it increases the synchronization delay to $2T$. After Maekawa's algorithm, many quorum-based mutual exclusion algorithms have been proposed to reduce the message complexity or the increase the resiliency to site and communication link failures. Since these algorithms are Maekawa-type algorithms, they also suffer from the long synchronization delay. In this paper, we propose a delay-optimal quorum-based mutual exclusion algorithm which reduces the synchronization delay to T and still has a low message complexity of $O(K)$ (K is the size of the quorum which can be as low as $\log N$). A correctness proof and a detailed performance analysis are provided.

Index Terms—Quorum, synchronization delay, distributed mutual exclusion, fault tolerance.

1 Introduction

In distributed systems, many applications, such as replicated data management, directory management, and distributed shared memory, require that a shared resource is allocated to a single process at a time. This is called the problem of mutual exclusion. The problem of mutual exclusion becomes much more complex in distributed systems (as compared to single-computer systems) due to the lack of both a shared memory and a common physical clock and because of unpredictable message delays.

Since a shared resource is expensive and processes that cannot get the shared resource must wait, the performance of mutual exclusion algorithms is critical to the design of high-performance distributed systems. The performance of mutual exclusion algorithms is generally measured by message complexity and synchronization delay. The message complexity is measured in terms of the number of messages exchanged per Critical Section (CS) execution. The synchronization delay is the time required after a site exits the CS and before the next site enters the CS and it is measured in terms of the average message delay T.

1.1 A Trade-Off Between Message Complexity and Delay

Over the last decade, many mutual exclusion algorithms [23] have been proposed to improve the performance of distributed systems, but they either reduce the message complexity at the cost of long synchronization delay or reduce the synchronization delay at the cost of high message complexity.

Lamport uses logical timestamp [10] to implement distributed mutual exclusion. For each CS execution, each site needs to get the permission from all other $(N-1)$ sites. The message complexity of this algorithm is $3*(N-1)$ and the synchronization delay is T. The Ricart-Agrawal algorithm [20] is an optimization of the Lamport algorithm. It reduces the *release* messages by cleverly merging them with the *reply* messages. This merging is achieved by deferring the lower priority requests. In this algorithm, the messages per CS execution are reduced to $2*(N-1)$ messages and the synchronization delay is still T. The dynamic algorithm in [22], on the average, requires $N-1$ messages per CS execution at light load and $2*(N-1)$ at heavy load, while the synchronization delay is still T. By exploiting the concurrency of requests and assigning multiple meanings to the requests and replies whenever there are concurrent requests, Lodha and Kshemkalyani [11] reduced the number of messages to somewhere between $N-1$ and $2(N-1)$, whereas the synchronization delay is still T.

The mutual exclusion algorithms in [7], [15], [19], on the average, require only $O(\log N)$ messages to execute the CS; however, the average delay in these algorithms increases to $O(\log N)$. The worst-case delay of the algorithm in [15] can be as much as $O(N)$. These algorithms have long delays because they impose some logical structure on the system topology (like a graph or tree) and a token request message must travel serially along the edges of the graph or tree.

In Maekawa's scheme [13], a set of sites called a quorum is associated with each site. The set (quorum) has a nonempty intersection with the sets corresponding to every other site. To enter the CS, a site only locks all sites in its quorum; thus, the message complexity is dramatically reduced. At light

- *G. Cao is with the Department of Computer Science and Engineering, Penn State University, University Park, PA 16802. E-mail: gcao@cse.psu.edu.*
- *M. Singhal is with the Department of Computer Science, University of Kentucky, KY 40506. E-mail: singhal@cs.uky.edu.*

Manuscript received 17 Mar. 2000; revised 29 Mar. 2001; accepted 4 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 111773.

load, a site needs to exchange $3 * (\sqrt{N} - 1)$ messages to achieve mutual exclusion. At heavy load, due to the requirement of handling deadlocks, the message complexity becomes $5 * (\sqrt{N} - 1)$. However, the synchronization delay increases $2T$ as opposed to T in other algorithms because a site exiting the CS must first send a *release* message to unlock the arbiter site which in turn sends a *reply* message to the next site to enter the CS (two serial message delays between the exit of the CS by a site and the entering into the CS by the next site).

1.2 Quorum Construction: A Trade-Off between Message Complexity and Availability

Recently, quorum-based mutual exclusion algorithms, which are a generalization of Maekawa's algorithm, have attracted considerable attention. Many algorithms [1], [4], [8], [9], [12], [14], [16], [17], [18] exist to construct quorums that can reduce the message complexity or increase the resiliency to site and communication failures.[1] In general, there is a trade-off between the message complexity and the degree of the resiliency of an algorithm. For example, majority voting [25] has high resiliency but relatively high message complexity $O(N)$, while the Maekawa algorithm has low message complexity $O(\sqrt{N})$ but relatively low resiliency to failures.

The tree algorithm [1] is based on organizing a set of N sites as nodes of a binary tree. A quorum is formed by including all sites along any path that starts at the root of the tree and terminates at a leaf. If a site in a path is unavailable, a quorum can still be formed by substituting that site with sites along a path, starting from a child node of the unavailable site to a leaf of the tree. The quorum size in the tree algorithm is $\log N$ in the best case and it becomes $\frac{N+1}{2}$ in the worst case. In the HQC or Hierarchical Voting Consensus algorithm [8], sites are organized in a multilevel hierarchy and voting is performed at each level of the hierarchy. The lowest level in the hierarchy contains groups of sites. In this construction, the quorum size becomes $N^{0.63}$. The Grid-set algorithm [4] has two levels. A majority voting scheme is used at the upper level to increase the resiliency, while a Maekawa-like grid structure is used at the lower level to reduce the message overhead. The quorum size is $\frac{N+1}{G} \sqrt{G}$, where G is the group size. The Rangarajan-Setia-Tripathi algorithm [18] in some sense is a dual of the Grid-set algorithm [4]. Specifically, they use majority voting at the lower (subgroup) level and a Maekawa-like grid structure at the higher level. This change reduces the quorum size to $\frac{G+1}{2} \sqrt{\frac{N}{G}}$, where G is the subgroup size.

1.3 Reducing the Synchronization Delay Algorithms

In addition to availability and message complexity, the synchronization delay needed for achieving quorum consensus is also recognized as an important factor. Especially for systems requiring short response time, such as replicated database systems, minimizing the delay for reaching consensus is a very important issue. After Maekawa published his famous paper [13] on quorum-based mutual exclusion, an open problem was to reduce the delay in processing users' requests from $2T$ to T. Many researchers worked on this problem.

Singhal uses the concept of mutable locks to achieve an optimal quorum-based algorithm [21] which is free from deadlocks and does not exchange messages like *inquire*, *fail*, and *yield* to resolve deadlocks. In this algorithm, the synchronization delay is reduced to T as opposed to $2T$ in Maekawa-type algorithms; however, the message complexity increases to $O(N)$.

Chang et al. [2], [3] proposed a hybrid approach to reduce the delay of quorum-based algorithms. In their approach, sites are divided into groups and different algorithms are used to resolve local (intragroup) and global (intergroup) conflicts. By carefully controlling the interaction between the local algorithm and the global algorithm, one can either minimize the message overhead (at the expense of increased delay) or minimize the delay (at the expense of increased message overhead). For example, they use Singhal's algorithm [22] as the local algorithm and Maekawa's algorithm as the global algorithm. By adjusting the parameters, at light traffic load, the algorithm simulates the performance of Singhal's algorithm, which has low delay and high message overhead. At heavy traffic load, the algorithm approaches Maekawa's algorithm, which has long delay and low message overhead.

In a quorum-based mutual exclusion scheme, the delay for reaching consensus depends critically on the constructed quorums and, thus, it is important to construct quorums with a small delay. Fu [5] introduced the notions of *max-delay* and *min-delay* of quorums. The max-delay of quorums is the maximum of the delays among all sites, while the mean-delay is the average. She has proposed polynomial-time algorithms to find max-delay optimal and mean-delay optimal quorums for systems with special topologies, such as trees and rings. Later, Tsuchiya et al. [26] extended this result to general graphs. Basically, these works [5], [26] concentrate on designing schemes that can construct delay-optimal quorums. They did not mention how to use these delay-optimal quorums. If Maekawa's scheme [13] is applied to achieve mutual exclusion, the synchronization delay is still $2T$, although the average message delay T in schemes using delay-optimal quorums may be smaller than that using nondelay-optimal quorums. Thus, more precisely, the schemes in [5] and [26] achieve optimal average message delay, which is different from achieving optimal synchronization delay.

In this paper, we solve the open problem of reducing the synchronization delay of quorum-based algorithms to T, while keeping the low message complexity $c * K$ (c is a constant between 3 and 6 and K is the average size of the quorum.) The basic idea is as follows: Instead of first sending a *release* message to unlock the arbiter site which in turn sends a *reply* message to the next site to enter the CS, the site exiting the CS directly sends a *reply* message to the site which enters the CS next. This reduces the

1. Early quorum-based algorithms are used to reduce the message complexity. Today, although network bandwidth has been improved, quorum-based algorithms still attract considerable attention due to their fault-tolerant property.

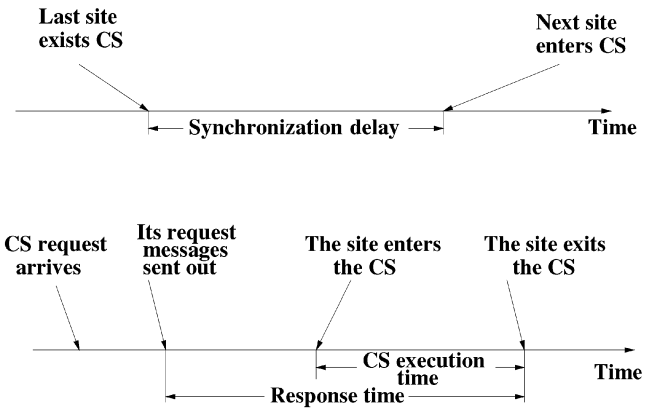


Fig. 1. Synchronization delay and response time.

synchronization delay from $2T$ to T. However, this change brings some complications and we discuss how to deal with them in this paper.

Our scheme is independent of the quorums being used. K is $\sqrt{N}$ if we use Maekawa's quorum construction algorithm [13] and K becomes $\log N$ when we use the Agrawal-Abbadi quorum construction algorithm [1]. Moreover, the redundancy in the quorum can increase the resiliency to site and communication link failures.

The rest of the paper is organized as follows: Section 2 develops the necessary background. In Section 3, we present the algorithm. The correctness proof and the performance analysis are provided in Section 4 and Section 5 respectively. In Section 6, we explain how to make this algorithm fault-tolerant. Section 7 concludes this paper.

2 Preliminaries

2.1 System Model and Definitions

A distributed system consists of N processes. The term *site* is used to refer to a process as well as the computer that the process is executing on. Sites are fully connected and they communicate with each other asynchronously by message passing. There are no global memory and no global clock. The underlying communication medium is reliable and sites do not crash. (If fault-tolerant quorum construction algorithms are used, our algorithm can handle site and communication failures.) The message propagation delay is unpredictable, but it has an upper bound and the messages between two sites are delivered in the order sent. A site executes its CS request sequentially one by one.

Synchronization Delay. Fig. 1 shows the definition of synchronization delay and response time. The *synchronization delay* is the time required after a site leaves the CS and before the next site enters the CS. Note that normally one or more sequential message exchanges are required after a site exists the CS and before the next site enters the CS. The *response time* is the time interval a request waits for its CS execution to be over after its request messages have been sent out. The *system throughput* is the rate at which the system executes requests for the CS. If $\mathcal{D}$ is the synchronization delay and E is the average critical section execution time, the throughput is given by the following equation:

$$\text{system throughput} = 1/(\mathcal{D} + E).$$

Quorum. Let U denote a nonempty set of N sites. A *coterie* $\mathcal{C}$ is a set of sets, where each set P in $\mathcal{C}$ is called a quorum. The following conditions hold for quorums in a coterie $\mathcal{C}$ under U [6]:

1. $(\forall P \in \mathcal{C} :: P \neq \emptyset \wedge P \subseteq U)$,
2. $Minimality\ Property: (\forall P, Q \in \mathcal{C} :: P \not\subset Q)$,
3. $Intersection\ Property: (\forall P, Q \in \mathcal{C} :: P \cap Q \neq \emptyset)$.

For example, $\mathcal{C} = \{\{\mathrm{a}, \mathrm{b}\}, \{\mathrm{b}, \mathrm{c}\}\}$ is a coterie under $U = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ and $P = \{\mathrm{a}, \mathrm{b}\}$ is a quorum.

The concept of intersecting quorum captures the essence of mutual exclusion in distributed systems. For example, to obtain mutually exclusive access to a shared resource in the system, a site, say S_i, is required to receive permission from all sites in the quorum of S_i. If all sites in the quorum of S_i grant the permission to S_i, S_i is allowed to access the shared resource. Since any pair of quorums have at least one site in common (by the Intersection Property), mutual exclusion is guaranteed. The Minimality Property is not necessary for correctness, but it is useful for efficiency.

2.2 The Basic Idea of Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms [13] associate a request set (quorum) R_i with a site S_i such that:

1. $(\forall i :: S_i \in R_i)$ and
2. $(\forall i \forall j :: R_i \cap R_j \neq \emptyset)$.

Site S_i executes its CS only after it has locked all the sites in R_i in exclusive mode. To do this, S_i sends *request* messages to all the sites in R_i. On receipt of the *request* message, site S_j immediately sends a *reply* message to S_i (indicating S_j has been locked by S_i) only if S_j is not locked by some other site at that time. Site S_i can access the CS only after receiving permission (i.e., *reply* messages) from all the sites in R_i. After having finished the CS execution, S_i sends *release* messages to all the sites in R_i to unlock them. Mutual exclusion is guaranteed because of the intersection property of quorums.

Quorum-based algorithms [13] are prone to deadlocks because a site is exclusively locked by other sites. Without loss of generality, assume three sites S_i, S_j, and S_k simultaneously invoke mutual exclusion. (Suppose $R_i \wedge R_j = \{S_{ij}\}, R_j \wedge R_k = \{S_{jk}\}$, and $R_k \wedge R_i = \{S_{ki}\}$.) Since sites do not send *request* messages to the sites in their request sets in any particular order, it is possible that, due to arbitrary message delay, S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}), S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}), and S_{ki} has been locked by S_k (forcing S_i to wait at S_{ki}). Thus, there is a waiting cycle: $S_j \rightarrow S_{ij} \rightarrow S_i \rightarrow S_{ki} \rightarrow S_k \rightarrow S_{jk} \rightarrow S_j$, which results in a deadlock.

Quorum-based algorithms [13] handle deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock (unless the former has succeeded in locking all the needed sites). A site suspects a deadlock

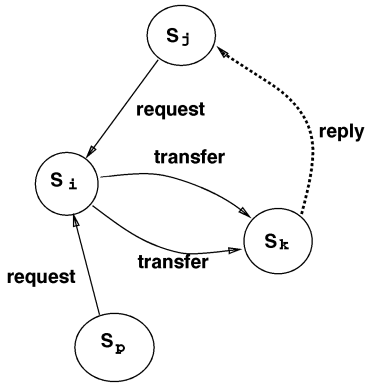


Fig. 2. Possible problems.

(and initiates message exchanges to resolve it) whenever a higher priority request finds that a lower priority request has already locked the site. Deadlock handling requires three new messages: *fail*, *inquire*, and *yield*, which will be explained in the next section.

3 A Delay-Optimal Quorum-Based Algorithm

In this section, we present our delay-optimal quorum-based mutual exclusion algorithm, which is independent of the quorums being used.

3.1 Main Issues

Our algorithm reduces the synchronization delay to T as follows: Instead of first sending a *release* message to unlock the arbiter site which in turn sends a *reply* message to the next site to enter the CS, the site exiting the CS directly sends a *reply* message to the site to enter the CS next. Although the idea may sound simple, its implementation is difficult and must address several issues. For example, how is a site informed about the next site to enter the CS? We solve this problem by using a *transfer* message as follows: In Fig. 2, S_i receives a *request* from S_j after S_i has sent a *reply* to S_k. On receipt of the *request*, S_i sends a *transfer* message to S_k to notify it that S_j is the next site to execute the CS. When S_k finishes its CS access, it sends a *reply* to S_j on behalf of S_i. When S_j receives the *reply*, it gets the permission to enter CS from S_i even though the *reply* was sent by S_k.

The exchange of *transfer* messages in this manner to inform a site about the next site to enter the CS may however result in the violation of mutual exclusion. Several scenarios can be constructed using Fig. 2, where mutual exclusion is violated:

- After S_i has sent a *transfer* message to S_k, S_i cannot send *reply* to any other sites until it knows that S_j has rejected the *reply* or S_j has finished its CS access. Otherwise, mutual exclusion will be violated. In order to avoid this problem, the *release* sent by S_k to S_i is modified to inform S_i if S_k has transfered a *reply* to other sites or not.
- If S_i receives S_p's *request* prior to S_k's *release* and S_p's *request* has higher priority than S_j's *request*, S_i sends another *transfer* to S_k to replace the previous *transfer*. If S_k responds to all *transfer* messages by sending corresponding *reply* messages, mutual exclusion may be violated. We solve this problem by making sure that S_k sends only one *reply* for the *transfer* messages from one sender.
- Assume S_i receives S_p's *request* after S_j's *request* and S_p's *request* has higher priority than S_j's *request*. In Maekawa's algorithm, S_i sends an *inquire* message to S_j to prevent deadlock. Maekawa assumes that a channel is $FIFO$. Consequently, an *inquire* message always arrives at a site later than the *reply* from the same sender. In our algorithm, a *reply* message from a site may come from different channels. For example, in Fig. 1, S_i's *reply* may come from S_i directly or from S_k indirectly by a *transfer*. Then, $FIFO$ assumption is not enough to ensure that an *inquire* arrives later than the *reply*. If this situation is not properly dealt with, it may result in a violation of mutual exclusion. In our algorithm, S_j defers responding to the *inquire* until it receives the *reply* from S_i, which is transfered by S_k.

3.2 Control Messages and Data Structures

Every request message is assigned a timestamp (the sequence number and the site number) according to Lamport's logical clock [10]. The sequence number assigned is greater than that of any request message sent, received, or observed at that site. The site with lower timestamp has higher priority which is determined as follows:

1. The message with smaller sequence number has higher priority.
2. If the messages have equal sequence numbers, the message with the smaller site number has higher priority.

Seven types of control messages are used in our scheme. The message format is as follows:

$$msg\_name(sender, receiver, additional\_parameters).$$

- ***request***: a $request(i, j, req_i)$ message from site S_i to site S_j indicates that S_i with request timestamp req_i (in the form of (sn, i)) is asking for S_j's permission to enter the CS.
- ***reply***: a $reply(i, j)$ message to site S_j indicates that S_i grants S_j's request to enter the CS.
- ***release***: A $release(i, j, req_k)$ message to S_j indicates that S_i has exited the CS. If $req_k \neq (max, max)$, S_i has transferred S_j's permission to site S_k.
- ***inquire***: An $inquire(i, j)$ message from S_i to S_j indicates that S_i wants to find out if S_j has succeeded in getting *reply* messages from all sites in R_j.
- ***fail***: A $fail(i, j)$ message from S_i to S_j indicates that S_i cannot grant S_j's request because it has currently granted the permission to a site with a higher priority request.
- ***yield***: A $yield(i, j)$ message from S_i to S_j indicates that S_i yields the right to enter the CS to a higher priority request and it is waiting for S_j's permission to enter the CS.

- ***transfer***: a $transfer(i, j, req_k)$ message from site S_i to site S_j indicates that S_i asks S_j to send a *reply* message to S_k on behalf of S_i after S_j exits the CS.

A site S_i maintains the following data structures:

- *lock*: a tuple (sn, j) (can also be represented by req_j) maintained by each site, where j is the site number of the request site to which S_i has granted a *reply* and sn is the sequence number of the request message. $lock$ is initialized to (max, max), where max is a number more than any site number and sequence number.
- *failed*: a Boolean which is initialized to zero each time a new CS request is sent. When S_i receives a *fail* or sends a *yield*, it sets $failed_i$ to 1.
- *replied*: a Boolean vector of size m (m is the size of quorum). The vector is initialized to zero each time a new CS request is sent. When S_i receives a $reply(j, i)$, it sets $replied_i[j]$ to 1.
- *req_q*: to queue the received *request* messages. Each entry in this queue is a tuple (sn, j), which is the timestamp of a *request*. The req_q is a priority queue (the *request* with the highest priority is on the top of the queue).
- *inq_set*: to save the $inquire$ messages which arrive at S_i earlier than the $reply$.
- *tran_stack*: to save all the *transfer* messages S_i receives. Every entry in this stack is a pair (j, req_k) which represents a $transfer(j, i, req_k)$ message. Due to out-of-order request messages, a site may receive multiple transfer messages. The receiver should only respond to the last transfer message. Thus, a stack is helpful. However, after responding to the last transfer message, other transfer messages from the same sender should be removed. Thus, an array of stacks should be implemented. For simplicity, we use a stack to describe the algorithm.

3.3 The Algorithm

To enter the CS, a site S_i requests the permission from each site in its quorum. If S_i has gotten the permission from all members in its quorum, it can enter the CS; otherwise, it continues to wait for the permission from the site which rejects its request.

When a site S_i, which has already been *locked* by S_k (S_i has sent a *reply* to S_k, but S_i has not received a *yield* or *release* from S_k), receives a *request* from S_j, S_i adds S_j's *request* into req_q_i. If S_j's *request* has the highest priority in req_q_i, S_i sends a *transfer* message to S_k, which forwards a *reply* message to S_j after it completes its CS execution. Note that, when S_j receives the forwarded *reply*, it gets permission to enter the CS from S_i even though the *reply* is not directly sent by S_i. S_i may send another *transfer* message to S_k in response to an out-of-order *request* message (i.e., a higher priority *request* arrives after a lower priority *request*). Upon exiting the CS, site S_k only sends a *reply* to the site whose *request* is the top entry in $tran\_stack_k$ and deletes the following entries in $tran\_stack_k$ from the same sender. This process is repeated until the $tran\_stack$ is empty. Since a site only sends a *transfer* to the site to which it has sent a *reply*, when a site S_k receives a *transfer* from another site, say S_i, $replied_k[i]$ should be equal to 1; otherwise, the *transfer* is an outdated *transfer* and should be discarded.

When site S_i receives a *release* from S_j, it first determines whether S_j has transferred a *reply* or not on its behalf based on the parameters of the *release*. If S_j has transferred a *reply* to a site called S_k, S_i saves S_k's *request* to $lock_i$ to reflect that S_k is locking S_i. If req_q_i is not empty, S_i sends a *transfer* to S_k based on the top entry in req_q_i. S_i sends a *reply* to the top entry site in req_q_i if S_j has not transfered the *reply*.

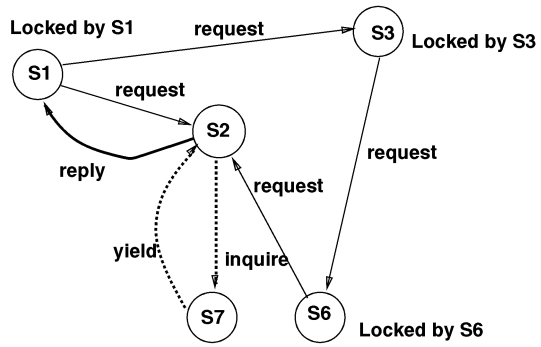
Since there is a danger of deadlock when more than one site simultaneously requests the CS, a site yields to another site if the priority of its request is lower than that of the other site. If a *request* with higher priority from S_j arrives at S_i and S_i has sent a *reply* to S_k, S_i sends an *inquire* message to S_k to inquire whether S_k has succeeded in getting the *reply* messages from all sites in its quorum. If S_k is unable to get *reply* messages from all sites in its quorum, e.g., it has sent a *yield* or it has received a *fail*, S_k returns a *yield* to S_j. Otherwise, it returns a *release* to S_i after it completes its CS execution. We use piggybacking to reduce the message complexity. For example, whenever a site sends an *inquire* in response to a higher priority *request*, the *inquire* is always piggybacked with a *transfer*.

If an *inquire* arrives earlier than the *reply* from the same sender, the receiving site defers responding to the *inquire* by putting it into inq_set. When a *reply* arrives, the receiver first checks if any *inquire* comes from the same sender as that of the *reply*. If so, process this *inquire*. If an *inquire* or *fail* from a site S_j arrives at S_i after S_i has sent *release* to S_j, S_i just ignores it.

In the formal description of our quorum-based mutual exclusion algorithm, in Action A.4, after dequeue req_q_i, a head(req_q_i) operation is applied, so we have to make sure that the req_q_i is not empty before the "head" operation. The reason is as follows: A site S_i receives a $yield(j, i)$ only when S_i has sent an $inquire$ to S_j. Also, S_i must have received a high priority request compared to S_j's $request$ and S_i has queued this request in its req_q_i (see Action 2). Then, after line 2 of Action 4, there are at least two items in req_q_i. After one dequeue operation, req_q_i is not empty.

A: Requesting the Critical Section:

1. /* For a site S_i wishes to enter CS */
   S_i sends $request(i, j, req_i)$ to every site $S_j \in R_i$;
   clear $tran\_stack_i$, inq_set_i, and $tran\_set_i$;
   /* $tran\_set$ is used to temporarily save transfer messages */
   $failed_i := 0$; $replied_i[] := 0$; $lock_i := (max, max)$;
2. Actions when S_i receives a $request(j, i, req_j)$:
   if $lock_i = (max, max)$
   then $lock_i := req_j$; send a $reply(i, j)$ message to S_j;
   else Let S_k is the site whose request is in $lock_i$;
   case $(req\_q_i = \emptyset) \wedge (req_j < lock_i)$:
   S_i sends $inquire(i, k)$ piggybacked with *transfer*(i, k, req_j) to S_k;
   case $(req\_q_i = \emptyset) \wedge (req_j > lock_i)$:
   S_i sends *transfer* (i, k, req_j) to S_k;
   S_i sends *fail(i, j)* to S_j;
   case $(req\_q_i \neq \emptyset) \wedge (req_j > head(req\_q_i))$
   S_i sends $fail(i, j)$ to S_j;



S1 = {1, 2, 3, 4}
S5 = {1, 5, 6, 7}
S8 = {1, 8, 9, 10}
S11 = {1, 11, 12, 13}
S2 = {2, 5, 8, 11}
S6 = {2, 6, 9, 12}
S7 = {2, 7, 10, 13}
S10 = {3, 5, 10, 12}
S3 = {3, 6, 8, 13}
S9 = {3, 7, 9, 11}
S13 = {4, 5, 9, 13}
S4 = {4, 6, 10, 11}
S12 = {4, 7, 8, 12}

Fig. 3. Handling possible deadlocks of Maekawa's algorithm.

case $(req\_q_i \neq \emptyset) \wedge (req_j < head(req\_q_i) < lock_i)$:
S_i sends *fail* to head (req_q_i);
S_i sends *transfer*(i, k, req_j) to S_k;
case $(req\_q_i \neq \emptyset) \wedge (req_j < lock_i < head(req\_q_i))$:
S_i sends *inquire(i, k)* piggybacked with *transfer*(i, k, req_j) to S_k;
case $(req\_q_i \neq \emptyset) \wedge (lock_i < req_j < head(req\_q_i))$:
S_i sends *transfer*(i, k, req_j) to S_k;
enqueue (req_q_i, req_j);

3. Actions when a site S_i receives an *inquire* (j, i):
if $(replied_i[j] = 1) \wedge (failed_i = 1)$
/* S_i has received a *fail* or sent a *yield**/
then $replied_i[j]$:= 0; $failed_i$:= 1;
send a $yield(i, j)$ to S_j;
delete all entries sent by S_j in $tran\_stack_i$;
else enqueue (inq_set_i, j);
4. Actions when a site S_i receives a $yield(j, i)$:
enqueue $(req\_q_i, lock_i)$; req_k := dequeue (req_q_i);
$lock_i$:= req_k; req_p := head(req_q_i);
send $reply(i, k)$ piggybacked with *transfer* (i, k, req_p) to S_k;
5. Actions when a site S_i receives a *transfer*(j, i, req_k):
if $reply_i[j] = 1$
then push $(tran\_stack_i, (j, req_k))$;
else ignore this *transfer*;
6. Actions when a site S_i receives a $reply(j, i)$:
$replied_i[j]$:= 1;
if $j \in inq\_set_i$
then delete j from inq_set_i;
Execute A.3 as if S_i receives $inquire(j, i)$;
7. Actions when a site S_i receives a *fail*(j, i):
$failed_i$:= 1;
for any $j \in inq\_set_i$
delete j from inq_set_i;
Execute A.3 as if S_i receives $inquire(j, i)$;

B: Executing the Critical Section:

A site S_i can accesses the CS only when for all S_k in R_i, $replied_i[k]$ = 1.

C: Releasing the Critical Section:

1. Actions when S_i exits the CS:
$replied_i[] = 0$;
while $tran\_stack_i \neq \emptyset$
$(k, req_j) := pop(tran\_stack_i)$;
S_i sends $reply(k, j)$ to S_j;
$tran\_set_i := tran\_set_i \cup (k, req_j)$;
delete other entries sent by S_k in $tran\_stack_i$;
For each $S_k \in R_i$:
if $\exists (k, req_j) \in tran\_set_i$:
/* there exists an entry sent by S_k in $tran\_set_i$ */
then send $release(i, k, req_j)$ to S_k;
else send $release(i, k, (max, max))$ to S_k;
2. Actions when a site S_i receives a $release(j, i, req_k)$:
if $req_k \neq (max, max)$
then if $req_k \notin req\_q_i$ exit from this Action;
$lock_i$:= req_k; delete req_k from req_q_i;
if $req\_q_i \neq \emptyset$
then req_p := head (req_q_i)
if $req_p < req_k$
then send $inquire(i, k)$ piggybacked with *transfer* (i, k, req_p) to S_k;
else send *transfer* (i, k, req_p) to S_k;
else if $req\_q_i = \emptyset$
then $lock_i$:= (max, max);
else req_p := dequeue (req_q_i); $lock_i$:= req_p;
if $req\_q_i = \emptyset$
then send $reply(i, p)$ to S_p;
else req_q := head (req_q_i);
send $reply(i, p)$ piggybacked with *transfer* (i, p, req_q) to S_p.

3.4 Handling Possible Deadlocks of Maekawa's Algorithm

Despite the exchange of *inquire*, *failed*, and *yield* messages, there is still a possibility of deadlocks in Maekawa's algorithm. Let us assume that all *request* messages in Fig. 3 have the same sequence number. S_1, S_3, S_6, S_7 all try to enter CS, therefore, they have obtained the *reply* messages from themselves. S_2 is in the quorum of S_1, S_6, S_7. S_7's *request* arrives at S_2 first, then S_6's, annd finally S_1's. Since S_2 has sent a *reply* to S_7, it sends an *inquire* to S_7 after S_6's *request* arrives at S_2. When S_1's *request* arrives, according to Maekawa's algorithm, S_2 continues to wait since it has sent an *inquire*. Assume S_6 has gotten all required *reply* messages except S_2's, S_3 has gotten all required *reply* messages except S_6's, and S_7 has been rejected by some other sites. As a result, S_2 sends *reply* to

S_1 after it receives the *yield* from S_7. However, S_3 cannot answer the *inquire* in response to S_1's *request* because it does not know whether it can lock all *reply* messages or not. For the same reason, S_6 cannot answer the *inquire* in response to S_3's *request*. There is a waiting cycle: $S_1 \rightarrow S_3 \rightarrow S_6 \rightarrow S_2 \rightarrow S_1$.

We avoid such deadlocks as follows: Suppose S_j's *request* is the top entry of req_q_i and S_k's *request* is in $lock_i$. When S_i receives a $request(p, i, req_p)$ which has higher priority than S_j's *request*, S_i compares the timestamp of S_p's *request* with that of S_k's *request*. If the priority of S_j's *request* is higher than that of S_k's *request*, S_i sends a *failed* to S_j. In this example, S_2 sends *failed* to S_6 when it receives S_1's *request* (The solution has been implemented in the proposed algorithm).

4 Correctness Proof

In this section, we show that the algorithm achieves mutual exclusion and is free from deadlock and starvation.

Assertion 1. *The proposed algorithm satisfies the following invariant:*

$$(I1)\ \ (\forall i \forall j (replied_i[j] = 1) \Longrightarrow (lock_j = req_i))$$
$$OR\ (I2)\ \ (\forall i \forall j (replied_i[j] = 1) \Longrightarrow (lock_j = req_k (k \neq i))$$
$$\wedge\ (reply_k[j] = 0) \wedge (S_i\ received\ reply(j,i)\ from\ S_k)).$$

(Invariant I1 states that if a site S_i obtains S_j's permission to enter the CS, then $lock_j = req_i$. Invariant I2 states that if a site S_i obtains S_j's permission to enter the CS, but $lock_j = req_k (k \neq i)$, then site S_k exits the CS and has transfered S_j's reply to S_i.)

Proof. The initial condition, where no site is requesting/executing CS and all $replied_i[]$ is zero, trivially implies the invariant. To show that the invariant is preserved by all the actions of the algorithm, we use the following standard technique: $I\{action\}\ C$ and show that post condition $C \Rightarrow I$.

We need to show that either *I1* or *I2* is preserved. From the algorithm, $replied_i[j]$ can only be set to one in Action A.6 as a result of receiving a $reply(j, i)$. There are two cases for a site S_i to receive $reply(j, i)$:

Case 1: S_j is the sender of the $reply(j, i)$. There are three actions (A.2, A.4, C.2) that S_j can send $reply(j, i)$ to S_i. Among all these actions, before sending $reply(j, i)$, $lock_j = req_i$ (the sender and receiver name used in the proof may be different from that in the algorithm and then result in different message parameters.) There are two possibilities when S_i receives $reply(j, i)$.

1. S_i does not yield S_j's *reply* to others before it exits the CS. In this case, $lock_j = req_i$ and *I1* is preserved before S_i exits the CS. After S_i exits the CS, $reply_i[j] = 0$ and, thus, invariant I (*I1* or *I2*) is preserved. Certainly, some other site may get S_j's *reply* after S_i exits the CS. Since S_i represents an arbitrary site, the proof does not lose any generality.
2. S_i yields S_j's *reply* to some other site. As a result (Action A.4), S_j may change its $lock_j$ and then $lock_j \neq req_i$. However, before sending $yield(i, j)$ to S_j, S_i must change its $replied_i[j]$ to zero (Action A.3). Thus, invariant I is preserved.

Case 2: $S_k (k \neq j)$ is the sender of the $reply(j, i)$, i.e., S_k transfers S_j's reply to S_i. This can only occur in Action C.1, where S_k exits from the CS and $reply_k[j] = 0$. For S_k to transfer S_j's $reply(j, i)$ to S_i, an entry (j, k, req_i) must be in the $tran\_stack_k$.

Note that a site only sends $transfer(j, k, req_i)$ to S_k when $lock_j = req_k$. $lock_j$ will not be changed before S_k exits the CS unless S_k yields S_j's reply. To yield S_j's reply (Action 3), S_k removes all entries sent by S_j in $tran\_stack_k$ and S_k cannot receive any transfer messages until $lock_j$ changes to req_k again. Since there is an entry (j, k, req_i) in $tran\_stack_k$, S_k must have received transfer messages from S_j and $lock_j = req_k$ when S_k transfers S_j's $reply(j, i)$ to S_i. There are two possibilities when S_i receives this $reply(j, i)$ from S_k.

1. S_i receives $reply(j, i)$ from S_k after S_j receives the $release(k, j, req_i)$. According to Action C.2, $req_i \neq (max, max)$, $lock_j$ is updated to req_i, and then *I1* is preserved. S_i may yield the reply, which has been discussed in Case 1.2. After S_i executes the CS, $replied_i[j] = 0$, invariant I is still preserved.
2. S_i receives $reply(j, i)$ from S_k before S_j receives the $release(k, j, req_i)$ from S_k. Before S_j receives the $release(k, j, req_i)$ from S_k, $lock_j$ cannot be updated since all messages, such as *inquire* and *transfer*, will be sent to S_k, which just ignores them (S_k has transfered the reply to S_i and exits the CS). Thus, *I2* is preserved before S_i exits from the CS. After S_i exits from the CS, $replied_i[j]$ becomes zero and invariant I is still preserved.

If S_j receives the $release(k, j, req_i)$ from S_k before S_i exits from the CS, the situation will be similar to Case 2.1. If S_j receives the $release(k, j, req_i)$ from S_k after S_i exits from the CS, $req_i \notin req\_q_j$ (Action C.2), this message will be ignored. Thus, invariant I will not be affected. □

Theorem 1. *Mutual exclusion is achieved.*

Proof. Assume to the contrary that two sites S_i and S_j are executing the CS simultaneously. From the Coterie Intersection Property: $(\forall P, Q \in \mathcal{C} :: P \cap Q \neq \emptyset)$, we know that S_i's quorum R_i and S_j's quorum R_j have at least one common site, say S_l. From Step B of the algorithm, if S_i and S_j are executing the CS simultaneously, both of them must have gotten S_l's permission; that is, $replied_i[l] = 1 \wedge replied_j[l] = 1$. Based on Assertion 1 and $replied_i[l] = 1$, we have

$$(I1)\ lock_l = req_i$$
$$OR\ (I2)\ (lock_l = req_k (k \neq i)) \wedge (replied_k[l] = 0)$$
$$\wedge\ (S_i\ \text{received}\ reply(l, i)\ \text{from}\ S_k).$$

Based on Assertion 1 and $replied_j[l] = 1$, we have

$$(I1')\ lock_l = req_j$$
$$OR\ (I2')\ (lock_l = req_k (k \neq j)) \wedge (replied_k[l] = 0)$$
$$\wedge\ (S_j\ \text{received}\ reply(l, j)\ \text{from}\ S_k).$$

It is easy to see that *I1* contradicts *I1′* and *I2′* and *I1′* contradicts *I2*. Based on Action C.1, a site exiting from the CS only responds one transfer message; that is, it only sends one reply message either to S_i or S_j, not both. Thus, *I2* contradicts *I2′*. A contradiction. □

Theorem 2. *A deadlock is impossible.*

Proof. Assume that a deadlock is possible. Then, none of the sites in a set of requesting sites is able to execute the CS because each of them is waiting for one or more *reply* messages. After a sufficient period of time, there must exist a waiting cycle among the sites requesting the CS. Every site is waiting for another one in the cycle.

In this cycle, there must exist a site S_i whose request has the highest priority. Suppose S_i is waiting for S_j's reply and S_j has sent a *reply* to S_k. According to algorithm A.2 and C.2, S_j sends an *inquire* to S_k.

1. Site S_k sends a *yield* to S_j. Then, S_j sends a *reply* to S_i according to A.3 and A.4. The cycle is broken.
2. Site S_k does not reply S_j's *inquire*. Then, S_k either enters the CS and breaks the cycle or waits for the reply of some other site S_p. Based on A.2 and A.3, S_p's *request* must have lower priority than S_k's *request*. Otherwise, S_k gets a *fail* and replies a *yield* according to A.2 and A.3. For the same reason, S_p must be waiting for the *reply* of a lower priority site. Otherwise, it enters the CS or sends a *yield* to break the cycle. The waiting chain continues to the site with the lowest priority. This site either enters the CS or sends a *yield* to the site waiting for its reply and breaks the cycle. A contradiction. □

Theorem 3. *Starvation is impossible.*

Proof. Starvation occurs when a site waits indefinitely to enter the CS while other sites are repeatedly entering and exiting the CS. Suppose there is a starving site S_i. From Theorem 2, there are always sites entering and exiting the CS. The starving site S_i must have sent *request* messages to all the sites in R_i and these *request* messages have arrived at the destination sites since communication channels are reliable. In our algorithm, any subsequent *request* is assigned a sequence number larger than all known sequence numbers. After a period of time, S_i's *request* has the highest priority among all the *request* messages received by each site in R_i. At that time, each site in R_i has sent a *reply* to S_i or has asked other sites to transfer a *reply* to S_i. Therefore, S_i receives all the *reply* messages and enters the CS in a finite time. A contradiction. □

5 A Performance Analysis

The performance of a mutual exclusion algorithm is often studied under two special loading conditions, i.e., *light load* and *heavy load*. In the analysis, a control message piggybacked with another message is counted as one message. The reason is as follows: The control message size is very small, but the message header is relatively large due to the requirements of the network protocols. Thus, the communication cost is mainly decided by the message header instead of the control message itself; that is, piggybacking one message with other control messages does not increase the communication cost significantly.

5.1 Message Complexity

5.1.1 Message Complexity under Light Load

Suppose the average quorum size is K. Under light load, the demand for the CS is low. Therefore, the contention for the CS is rare and the execution of the CS requires $(K-1)$ *request*, $(K-1)$ *reply*, and $(K-1)$ *release* messages, resulting in $3(K-1)$ messages per CS execution.

5.1.2 Message Complexity under Heavy Load

Suppose site S_i receives a $request(j, i, req_j)$ from S_i after S_i has sent a *reply* to S_k. When the demand is heavy, there are several situations to consider:

Case 1 $(req\_q_i = \emptyset) \wedge (req_j > lock_i)$. The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *fail*, $(K-1)$ *transfer*, $(K-1)$ *reply*, and $(K-1)$ *release* messages, which results in $5(K-1)$ messages.

Case 2 $(req\_q_i = \emptyset) \wedge (req_j < lock_i)$ *OR* $(req\_q_i \neq \emptyset) \wedge (req_j < lock_i < head(req\_q_i))$. There are two cases depending on whether the inquired site has replied a *yield* or not.

1. (has not replied a *yield*): The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *inquire* piggybacked with *transfer*, $(K-1)$ *reply*, $(K-1)$ *release* messages, and $(K-1)$ *transfer* messages, which results in $5(K-1)$ messages to enter the CS.
2. (has replied a *yield*): The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *inquire* piggybacked with *transfer*, $(K-1)$ *yield*, $(K-1)$ *reply* piggybacked with *transfer*, and $(K-1)$ *release* messages, which results in $5(K-1)$ messages per CS execution.

Case 3 $(req\_q_i \neq \emptyset) \wedge (req_j > head(req\_q_i))$. The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *fail*, $(K-1)$ *reply*, $(K-1)$ *release*, and $(K-1)$ *transfer* messages, which results in $5(K-1)$ messages.

Case 4 $(req\_q_i \neq \emptyset) \wedge (req_j < head(req\_q_i) < lock_i)$. There are two cases depending on whether the inquired site has replied a *yield* or not.

1. (has not replied a *yield*): The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *fail*, $(K-1)$ *transfer*, $(K-1)$ *release*, and $(K-1)$ *reply* messages, which results in $5(K-1)$ messages per CS execution.
2. (has replied a *yield*): The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *fail*, $(K-1)$ *transfer*, $(K-1)$ *yield*, $(K-1)$ *reply* piggybacked with *transfer*, and $(K-1)$ *release* messages, which results in $6(K-1)$ messages per CS execution.

Case 5 $(req\_q_i \neq \emptyset) \wedge (lock_i < req_j < head(req\_q_i))$. The execution of a CS requires $(K-1)$ *request*, $(K-1)$ *transfer*, $(K-1)$ *release*, $(K-1)$ *reply*, and $(K-1)$ *transfer* messages, which results in $5(K-1)$ messages per CS execution.

Based on this analysis, the proposed algorithm requires $5(K-1)$ or $6(K-1)$ messages per CS access under heavy load. Note that, only in Case 4.2, our algorithm requires $6(K-1)$ messages per CS access.

5.2 Synchronization Delay

The synchronization delay under light load becomes meaningless because it depends on the interrequest arrival time. The response time under light load is $2T + E$ (E is the CS execution time), which is necessary for any mutual exclusion algorithms under light traffic load.

5.2.1 Synchronization Delay when $E \geq T$

When site S_i receives a high priority *request* from S_j after it has granted its *reply* to a low priority site, say S_k, it starts a deadlock resolution; that is, S_i sends an *inquire* to S_k. Eventually, S_k sends a *yield* or a *release* to S_i in response to the *inquire*. As a result, S_i sends a *reply* to S_j. The deadlock resolution is finished when S_j receives the *reply* from S_i.

Theorem 4. *Suppose that a site S_i is executing the CS and a site S_j is the next site to enter the CS. If S_j has finished all deadlock resolutions before it receives a reply or transferred reply from S_i, the synchronization delay is T when $E \geq T$.*

Proof. If S_j is waiting for the reply from a site, say S_k, there are two possible situations:

Case 1 ($k = i$). After S_i exits the CS, it sends a *reply* to S_j and then S_j enters the CS. Thus, the synchronization delay is T.

Case 2 ($k \neq i$). There are two possibilities.

1. S_i is locking S_k's reply. In this case, S_k sends a $transfer(k, i, req_j)$ to S_i. After S_i exits the CS, it sends a *reply* message to the S_j on behalf of S_k directly. Thus, the synchronization delay is T.
2. $S_p (p \neq i)$ is locking S_k's reply. S_j's request must have higher priority than S_p's request; otherwise, S_j cannot be the next site to enter the CS since S_p enters the CS before S_j. If S_j's request has higher priority than S_p's request, a deadlock resolution is invoked. According to our assumption, the deadlock resolution is finished before S_j receives a *reply* or a transferred *reply* from S_i; that is, when S_j receives a *reply* or a transferred *reply* from S_i, S_j has received the *reply* from S_k even though S_k was locked by S_p before the deadlock resolution. Based on the result of Case 1 and Case 2.1, the synchronization delay is T. □

Under heavy load,[2] a site needs to wait for a long time to enter the CS. For simplicity, we assume that deadlock resolution is much faster than the waiting time under heavy load because a deadlock resolution starts as soon as the *request* messages arrive but the waiting time under heavy load may be long. The validity of this assumption depends on implementation details and environment conditions. Based on this assumption, during the long wait, the deadlock resolution can be finished before a site receives a *reply* or a transferred *reply* from the last site executing the CS. Thus, based on Theorem 4, the synchronization delay is T.

Note that in the proof of Theorem 4, if S_j's deadlock resolution has not been finished when it receives a *reply* or a transferred *reply* from S_i, the synchronization delay may be longer than T. This can be explained by the following example. In Fig. 4, S_i is executing the CS and S_j is the next site to enter the CS. S_j is waiting for the *reply* from site S_k and S_k's *reply* is locked by S_p. S_j's request has higher priority than S_p's request; hence, S_k sends an *inquire* to S_p to start a deadlock resolution. Eventually, S_p receives a *fail* message; otherwise, S_j cannot be the next site to enter the CS since S_p enters the CS before it. This *fail* should be received when S_p receives the *inquire* from S_k since S_p needs $2T$ to get the *reply* from S_k and it also needs $2T$ to get a *fail* (assume message delays are equal). Thus, S_p sends a *yield* to S_k which in turn sends a *reply* to S_j (not shown in the figure). Suppose that S_p sends its *yield* to S_k when S_i exits the CS, the synchronization delay is $2T$.

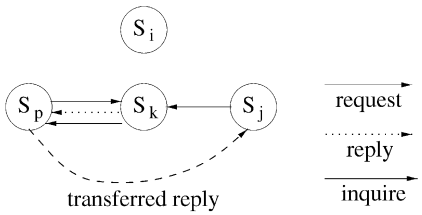


Fig. 4. Further reducing the synchronization delay.

Based on the idea of *transfer* messages used in our algorithm, we can reduce the synchronization delay to T as follows: When S_k sends the *inquire* to S_p, it asks S_p to transfer a *reply* to S_j. In response to the *inquire*, S_p sends a *reply* to S_j, on behalf of S_k, and then sends a *yield* or a *release* to S_k, which in turn updates its *lock*. As a result, S_j only takes time T to receive the *reply* from S_k even though the *reply* is sent by S_p. Thus, the synchronization delay is still T and the piggybacked *transfer* (with *inquire*) is not needed. In the worst case, S_j may send its *request* to S_k when S_i exits the CS, resulting in a synchronization delay of $3T$. However, in this case, quorum-based algorithms has a delay of $4T$ and other algorithms, such as Lamport's algorithm [10], Ricart-Agrawal algorithm [20], etc., need $2T$. Hence, we consider this special case as a light load.

During the analysis, we assume that the message delays are equal. If message delays are not equal, the synchronization delay in our algorithm may be longer than (some) T when the finish time of deadlock resolutions does not satisfy the requirement of Theorem 4. For example, in Fig. 4, S_p may not have gotten a *yield* when it receives the *inquire* from S_k since message delays may be different, the delay of $2T$ to get the *yield* may be longer than the $2T$ to get the *reply*. As a result, the synchronization delay is a little bit longer than the T (message delay from S_p to S_j). In algorithms such as Lamport's algorithm [10], Ricart-Agrawal algorithm [20], etc., the T is the longest message delay (not the average message delay) in the system. However, the T in our algorithm may be significantly shorter than the longest message delay. For example, due to the redundancy of fault-tolerant quorums [4], [18], a site does not need to receive all replies from all the sites in the fault-tolerant quorum. Instead, it only needs to receive the replies from

2. "Under heavy load" does not necessarily mean that the whole system is under heavy load. It may occur when only one arbiter site is under heavy load.

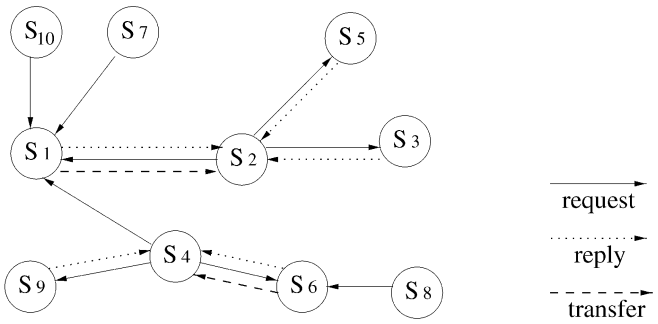


Fig. 5. Synchronization delay when $E < T$.

some of the sites in a quorum depending on the redundancy of the quorum. As a result, when a fault-tolerant quorum is used, our algorithm may not need to wait for those long delayed messages.

According to [21], deadlocks cannot be avoided without increasing the message complexity to $O(N)$. Thus, deadlock resolution is necessary in order to keep the advantage of low message complexity of the quorum-based mutual exclusion algorithms. From Theorem 4, without considering the finish time of deadlock resolutions, our algorithm has an optimal synchronization delay of T. The proposed algorithm also has the lowest synchronization delay when the finish time of deadlock resolutions is considered, since the deadlock resolution delay is necessary for any low message complexity quorum-based algorithms. In the following sections, we assume that the finish time of deadlock resolutions satisfies the requirement of Theorem 4, which is a typical situation under heavy load.

5.2.2 Synchronization Delay when $E < T$

When $E < T$, the synchronization delay may be longer than T. This can be explained by the following example. In Fig. 5, after S_1 sends a *reply* to S_2, it receives a request from S_4 and then it sends a *transfer* to S_2. Suppose S_4's request has higher priority than S_7's request. S_1 does not send a *transfer* to S_2 after it receives the request from S_7. There are four sites in S_4's quorum: $\{S_1, S_4, S_6, S_9\}$ and S_4 has gotten the permission from all sites in its quorum except S_1 which is locked by S_2. Suppose S_4's request has higher priority than S_8's request. When S_6 receives the request from S_8, it sends a *transfer* to S_4. After S_2 gets the permission from all sites in its quorum ($\{S_1, S_2, S_3, S_5\}$), it enters the CS. When S_2 exits the CS, it sends a *reply* to S_4 on behalf of S_1. As a result, S_4 enters the CS and the synchronization delay is T. When S_2 exits the CS, it also sends a *release* to S_1 to notify S_1 that S_2 has sent a *reply* to S_4 on behalf of S_1. As a result, S_1 sends $transfer(1, 4, req_7)$ to S_4 so that S_4 can send a *reply* to S_7 after it exits the CS. However, the *release* message needs time T to reach S_1 and another T for the *transfer* to arrive at S_4. If $E < T$, S_4 exits the CS before it receives the *transfer* from S_1. If S_7 is the next site to enter the CS (S_7 has gotten the permission from all sites in its quourm), the synchronization delay will be $2T$,[3] since it takes time T for S_1 to receive the *release* from S_4 and another T for its *reply* to arrive at S_7. If some other site, say S_8, which has received all necessary *transfer* messages, is the next site to enter the CS after S_4, the synchronization delay is still T. Note that, even if S_7 is the next site to enter the CS, the synchronization delay is $2T$ only for this situation. After S_7 exits the CS, no matter which site enters the CS next, the synchronization delay will be T since all *transfer* messages have enough time to reach the site which enters the CS next.

Suppose S_i is locking S_j's *reply* and S_i exits the CS before it receives a *transfer* from S_j. Suppose the next site to enter the CS is S_k. Let p denote the probability of S_j in S_k's quorum. The synchronization delay $\mathcal{D}$ is given by the following expression:

$$\mathcal{D} = \begin{cases} T & \text{if } E \geq T \\ \frac{T+(1-p)*T+p*2*T}{2} & \text{if } E < T. \end{cases} \quad (1)$$

If we use Maekawa's quorum construction algorithm [13], and the traffic is uniformly distributed, then $p = \frac{1}{\sqrt{N}}$. However, if all sites that cannot enter the CS are waiting for the same arbiter site (the worst case), $p = 1$ and $\mathcal{D} = \frac{3*T}{2}$. In the following, we modify our algorithm and reduce the synchronization delay to T even when $E < T$.

5.3 An Enhancement

When $E < T$, the synchronization delay increases since *transfer* messages may not arrive at the site executing the CS in time. To solve the problem, we allow *transfer* messages to be sent to a site which is not locking the arbiter's *reply*; that is, we allow the *transfer* to be sent one step ahead. For example, in Fig. 5, when S_1 receives S_4's *request*, it sends $transfer(1, 2, req_4)$ to S_2. When S_1 receives S_7's *request*, since it has sent a *transfer* to S_2, it sends $transfer(1, 4, req_7)$ to S_4 although S_4 is not locking S_1's *reply*. As a result, after S_4 exits the CS, it sends a *reply* to S_7 on behalf of S_1, resulting in a synchronization delay of T instead of $2T$ in the previous example. Note that, if S_2 is locking S_1's *reply* when S_{10}'s *request* arrives, S_1 does not send a *transfer* until it receives a *release* from S_2. When S_1 receives the *release*, it sends $transfer(1, 7, req_{10})$ to S_7. In summary, the arbiter site sends *transfer* messages one step ahead.

The above approach does not increase any message overhead since the *transfer* messages are only sent one step ahead. However, message overhead is increased when a *request* arrives out-of-order, where a new *transfer* needs to be sent again. (This also applies to the algorithm presented in Section 3.) To save message overhead, when a site S_i receives several *transfer* messages from the arbiter site S_j, it sorts them according to their *request* priority and only sends a *reply* to the site, say S_k, with the highest priority. Also, it piggybacks other *transfer* messages with the transferred *reply*. When S_i sends a *release* to the arbiter site S_j, it notifies S_j about the piggybacked *transfer* messages that it has sent to S_k. As a result, S_j does not need to send a *transfer* message until the piggybacked *transfer* is empty (based on the information piggybacked in the *release*).

With this enhancement, the synchronization delay is reduced to T without increasing message overhead, even when $E < T$. Since the site that exits the CS needs at least one message delay to notify the next site to enter the CS, the

3. Techniques exist to reduce the synchronization delay to $2T - E$. We do not disscuss it since we will discuss an enhancement in the next section that reduces the delay to T.

TABLE 1
A Comparison of Performance ($ll =$ **light load**, $hl =$ **heavy load**)

NON-TOKEN	Sync Delay(hl)	Messages(ll)	Messages(hl)
Lamport	T	$3(N-1)$	$3(N-1)$
Ricart-Agrawal	T	$2(N-1)$	$2(N-1)$
Singhal	T	$3(N-1)/2$	$3(N-1)/2$
Maekawa	$2T$	$3(\sqrt{N}-1)$	$5(\sqrt{N}-1)$
Our algorithm($K=\sqrt{N}$)	T	$3(\sqrt{N}-1)$	$6(\sqrt{N}-1)$
Our algorithm($K=\log N$)	T	$3(\log N-1)$	$6(\log N-1)$
TOKEN	Sync Delay	Messages(ll)	Messages(hl)
Suzuki-Kasami [24]	T	N	N
Singhal's heuristic	T	$N/2$	N
Raymond	$T(\log N)/2$	$\log N$	4

minimum synchronization delay is T. Thus, our algorithm is a delay-optimal quorum-based mutual exclusion algorithm.

Correctness. To simplify the proof, we only give the sketch of the proof based on the correctness proof in Section 4. The enhancement allows the *transfer* message to be sent one step ahead. Since it does not violate any conditions and modifies control messages, such as *inquire*, *reply*, *request*, *fail*, and *yield*, which are used in the proof of Theorem 2 and Theorem 3, the deadlock impossibility proof and the starvation impossibility proof are similar to Theorem 2 and Theorem 3. The assertions $I1$ and $I2$ are still valid since sending the *transfer* message only helps some processes get the *reply* earlier and it does not change the condition of sending a *reply*, which is the major focus in the proof of assertions $I1$ and $I2$. Since assertions $I1$ and $I2$ are still valid and the enhancement does not change Action C.1 of the algorithm, the property of mutual exclusion can be proven similar to Theorem 1. Similarly, piggybacking *transfer* messages with the transfered $reply$ does not affect the correctness since we can look at it as another way of sending *transfer* messages.

5.4 Comparison with Other Algorithms

The proposed algorithm is independent of the type of quorum being used. K becomes $\sqrt{N}$ if we use Maekawa's quorum construction algorithm [13] and K is $\log N$ when we use the Agrawal-Abbadi quorum construction algorithm [1]. Table 1 shows the message complexity and the synchronization delay for the proposed and various existing mutual exclusion algorithms. We observe that our algorithm has the lowest synchronization delay and still has a low message complexity. Although Raymond's algorithm has lower message complexity, it has long synchronization delay and suffers from the token loss problem.

6 Adding Fault Tolerance

Many quorum-based algorithms [1], [4], [8], [9], [12], [13], [16], [17], [18] have been proposed for mutual exclusion in distributed system. In general, there is a trade-off between the message complexity and the degree of the resiliency of an algorithm. For example, majority voting [25], which has high resiliency, has relatively high message complexity $O(N)$, whereas Maekawa's algorithm, which has low message complexity $O(\sqrt{N})$, has relatively low resiliency to failures. Much progress has been made to increase the resiliency of mutual exclusion algorithms. For example, The tree algorithm [1] is based on organizing a set of N sites as nodes of a binary tree. A quorum is formed by including all sites along any path that starts at the root and terminates at a leaf. If a site in a path is unavailable, a quorum can still be formed by substituting that site with sites along a path starting from a child node of the unavailable site to a leaf of the tree. Other algorithms such as the Hierarchical Voting Consensus algorithm [8], the Grid-set algorithm [4], and the Rangarajan-Setia-Tripathi algorithm [18] can also construct fault tolerant quorums.

If our algorithm uses the fault tolerant quorum constructed by any of these algorithms [1], [4], [8], [18], it becomes a fault tolerant mutual exclusion algorithm. Since all these quorums satisfy the intersection property, the correctness of the algorithm is maintained.

There is a difference between the Rangarajan-Setia-Tripathi algorithm [18] (or the Grid-set [4]) and the tree algorithm [1] (or the HQC algorithm [8]). When a site fails, the former can tolerate the failure without any recovery scheme (this is achieved by majority voting in the subgroup), but the latter needs a recovery scheme because a new quorum must be constructed. Note that, even in the former, a recovery scheme increases the failure resiliency. We enhance our mutual exclusion algorithm in the following way to make it resilient to failures.

When a site finds out that a site, say S_i, has failed, it broadcasts (based on known quorum information, multicast is enough) a $failure(i)$ message. A site, say S_j, on receiving a $failure(i)$ message, acts as follows:

1. S_j checks whether $S_i \in R_j$. If so, it makes S_i inaccessible, releases all the resources it has gotten, and executes the quorum construction algorithm to select another quorum.
2. S_j checks whether S_i's *request* (req_i) is in its req_q_j, $tran\_stack_j$, or $lock_j$:

Case 1 $req_i \in req\_q_j$. If req_i is the top entry in req_q_j and req_q_j has more than one entry, S_j deletes req_i from req_q_j and sends a new $transfer$ to the site in $lock_j$. Otherwise, S_j just deletes req_i from req_q_j.

Case 2 $req_i \in tran\_stack_j$. Delete req_i from $tran\_stack_j$;

Case 3 $req_i \in lock_j$. In this case, S_i is locking S_j. Therefore, S_j releases itself from S_i, and sends a *reply* piggybacked with a *transfer* to the site whose *request* is the top entry in req_q_j. The formal description is as follows:

if $req\_q_j = \emptyset$
then $lock_j := (max, max)$;
else req_p := dequeue(req_q_j); $lock_j := req_p$;
  if $req\_q_j = \emptyset$
  then send $reply(j, p)$ to S_p;
  else req_q := head(req_q_j);
    send $reply(j, p)$ piggybacked with
    transfer(j, p, req_q) to S_p.

7 Conclusions

Quorum-based mutual exclusion is an attractive approach for providing mutual exclusion in distributed systems due to its low message complexity and high resiliency. After the first quorum-based algorithm [13] was proposed by Maekawa more than a decade ago, many algorithms [1], [4], [8], [9], [12], [16], [17], [18] have been proposed to construct different quorums to reduce the message complexity or increase the resiliency to site and communication failures. Some researchers also propose schemes for constructing delay-optimal quorums to reduce the average message delay. However, all these quorum-based algorithms depend on Maekawa's algorithm to ensure mutual exclusion and they all have high synchronization delay ($2T$).

In this paper, we proposed a quorum-based mutual exclusion algorithm which reduces the synchronization delay to T and still has the low message complexity of $O(K)$ (K is the size of the quorum, which can be as low as $\log N$). This has two very beneficial implications: First, at heavy loads, the rate of CS execution (i.e., throughput) can almost be doubled. Second, at heavy loads, the waiting time of requests can be reduced to half because the CS executions proceed with twice the rate. Our algorithm is independent of the quorums being used. By using a fault-tolerant quorum, the algorithm increases the resiliency to site and communication failures. Even though we mainly discussed mutual exclusion in this paper, the proposed idea can be used in replicated data management, as long as the quorum being used supports replica control.

Acknowledgments

The authors would like to thank the anonymous referees whose insightful comments helped us to improve the presentation of the paper. This work was supported in part by the US National Science Foundation CAREER Award CCR-0092770.

References

[1] D. Agrawal and A.E. Abbadi, "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion," *ACM Trans. Computer Systems,* Feb. 1991.
[2] Y.-I. Chang, "A Simulation Study on Distributed Mutual Exclusion," *J. Parallel and Distributed Computing,* vol. 33, pp. 107-121, 1996.
[3] Y.-I. Chang, M. Singhal, and M. Liu, "A Hybrid Approach to Mutual Exclusion for Distributed Systems," *Proc. Ann. Int'l Computer Software and Application Conf.,* pp. 289-294, Oct. 1990.
[4] S.Y. Cheung, M.H. Ammar, and M. Ahamad, "The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data," *IEEE Trans. Knowledge and Data Eng.,* June 1992.
[5] A. Fu, "Delay-Optimal Quorum Consensus for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, Jan. 1997.
[6] H. Garcia and D. Barbara, "How to Assign Votes in a Distributed System," *J. ACM,* May 1985.
[7] J. Helary, A. Mostefaoui, and M. Raynal, "A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 11, pp. 1185-1196, Nov. 1994.
[8] A. Kumar, "Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data," *IEEE Trans. Computers.* pp. 996-1004, Sept. 1991.
[9] Y.-C. Kuo and S.-T. Huang, "A Geometric Approach for Constructing Coteries and k-Coteries," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 4, pp. 402-411, Apr. 1997.
[10] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[11] S. Lodha and A. Kshemkalyani, "A Fair Distributed Mutual Exclusion Algorithm," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 6, pp. 537-549, June 2000.
[12] W.-S. Luk and T.-T. Wong, "Two New Quorum Based Algorithms for Distributed Mutual Exclusion," *Proc. 17th Int'l Conf. Distributed Computing Systems,* May 1997.
[13] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Trans. Computer Systems,* May 1985.
[14] D. Malkhi and M. Reiter, "An Architecture for Survivable Coordination in Large Distributed Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 2, pp. 187-202, Mar./Apr. 2000.
[15] M. Naimi and M. Trehel, "An Improvement of the Log(n) Distributed Algorithm for Mutual Exclusion," *Proc. Seventh Int'l Conf. Distributed Computing System,* pp. 371-375, 1987.
[16] W.K. Ng and C.V. Ravishankar, "Coterie Templates: A New Quorum Construction Method," *Proc. 15th Int'l Conf. Distributed Computing Systems,* pp. 92-99, May 1995.
[17] D. Peleg and A. Wool, "Crumbling Walls: A Class of Practical and Efficient Quorum Systems," *Distributed Computing,* vol. 10, no. 2, pp. 120-129, 1997.
[18] S. Rangarajan, S. Setia, and S.K. Tripathi, "A Fault-Tolerant Algorithm for Replicated Data Management," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 12, pp. 1271-1282, Dec. 1995.
[19] K. Raymond, "A Tree-Based Algorithm for Distributed Mutual Exclusion," *ACM Trans. Computing Systems,* pp. 61-77, Feb. 1989.
[20] G. Ricart and A.K. Agrawal, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM,* vol. 24, no. 1, Jan. 1981.
[21] M. Singhal, "A Class of Deadlock-Free Maekawa-Type Algorithms for Mutual Exclusion in Distributed Systems," *Distributed Computing,* vol. 4, pp. 131-138, Feb. 1991.
[22] M. Singhal, "A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* Jan. 1992.
[23] M. Singhal, "A Taxonomy of Distributed Mutual Exclusion," *J. Parallel and Distributed Computing,* vol. 18, pp. 94-101, May 1993.
[24] I. Suzuki and T. Kasami, "A Distributed Mutual Exclusion Algorithm," *ACM Trans. Computer Systems,* 1985.
[25] T.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. Database Systems,* pp. 180-209, June 1979.
[26] T. Tsuchiya, M. Yamaguchi, and T. Kikuno, "Minimizing the Maximum Delay for Reaching Consensus in Quorum-Based Mutual Exclusion Schemes," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 4, Apr. 1999.

Guohong Cao received the BS degree from Xian Jiaotong University, Xian, China. He received the MS and PhD degrees in computer science from the Ohio State University in 1997 and 1999, respectively. Since Fall 1999, he has been an assistant professor of computer science and engineering at Pennsylvania State University. His research interests include distributed fault-tolerant computing, mobile computing, and wireless networks. He was a recipient of the Presidential Fellowship at the Ohio State University. He is a recipient of the US National Science Foundation CAREER award. He is a member of the IEEE and the IEEE Computer Society.

Mukesh Singhal received the BEng degree in electronics and communication engineering with high distinction from the University of Roorkee, Roorkee, India, in 1980 and the PhD degree in computer science from University of Maryland, College Park, in May 1986. He is a full professor of computer and information science at The Ohio State University, Columbus, Ohio. received the BEng degree in electronics and communication engineering with high distinction from the University of Roorkee, Roorkee, India, in 1980 and the PhD degree in computer science from University of Maryland, College Park, in May 1986. His current research interests include operating systems, database systems, distributed systems, performance modeling, mobile computing, computer networks, and computer security. He has published more than 140 refereed articles in these areas. He coauthored *Advanced Concepts in Operating Systems* (McGraw-Hill, 1994) and *Readings in Distributed Computing Systems* (IEEE Computer Society Press, 1993). He is a fellow of IEEE. He is currently serving in the editorial board of *IEEE Transactions on Knowledge and Data Engineering* and *Computer Networks*. He is currently serving as the program director of the Operating Systems and Compilers program at the US National Science Foundation. He is a member of the IEEE Computer Society and a fellow of the IEEE.