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

Guohong Cao and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
E-mail: {gcao, singhal}@cis.ohio-state.edu

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. Lamport's algorithm and 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 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 increase the resiliency to site and communication link failures. Since these algorithms are Maekawa-type algorithms, they also suffer from long synchronization delay $2T$. 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.

Key words: Quorum, synchronization delay, distributed mutual exclusion, fault-tolerance.

*This is a substantially enhanced version of a paper presented at ICDCS'98.

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) because of the lack of both a shared memory and a common physical clock and due to 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.

Over the last decade, many mutual exclusion algorithms [20] 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 message complexity.

Lamport uses logical timestamp [7] 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.

Ricart-Agrawal algorithm [16] is an optimization of Lamport's algorithm that reduces the *release* messages by cleverly merging them with *reply* messages. This merging is achieved by deferring the lower priority requests. In this algorithm, the messages per CS execution is reduced to $2*(N-1)$ messages and the synchronization delay is still T. The dynamic algorithm in [19] on the average requires $N-1$ messages per CS execution at light load and $2*(N-1)$ at heavy load. The synchronization delay is still T.

In Maekawa's scheme [9], a set of sites called a quorum is associated with each site, and this set has a nonempty intersection with the sets corresponding to every other sites. To enter the CS, a site only locks all sites in its quorum; thus, the message complexity is dramatically reduced. At light load, a site needs to exchange $3*(\sqrt{N}-1)$ messages to achieve mutual exclusion. At heavy load, because of the need to handle deadlocks, the message complexity becomes $5 * (\sqrt{N} - 1)$. However, the synchronization delay becomes $2T$ 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 enter into the CS by the next site).

Singhal uses the concepts of mutable locks to achieve an optimal deadlock-free Maekawa-type algorithm [18] 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 $O(N)$.

In Singhal's token-based heuristic algorithm [17], each site maintains information about the state of other sites in the system and uses it to select a set of sites that are likely to have the token. The site requests the token only from these sites, reducing the number of messages required to execute the CS. Although the synchronization delay is T, the message complexity varies between 0 and N.

The mutual exclusion algorithms in [4, 10, 15] on the average require only $\mathrm{O}(\log N)$ messages to execute the CS; however, the average delay in these algorithms is also $\mathrm{O}(\log N)$. The worst case delay of the algorithm in [10] can be as much as $\mathrm{O}(N)$. 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. Besides the long delay, token-based algorithms suffer from token loss problems [1].

Recently, quorum-based mutual exclusion algorithms, which are a generalization of Maekawa's algorithm, have attracted considerable attention. Many algorithms [1, 2, 5, 6, 8, 9, 11, 12, 13, 14] exist to construct quorums that can reduce the message complexity or increase the resiliency to site and communication failures. However, not much work has been done on minimizing the synchronization delay. Because all quorum-based algorithms are Maekawa-type algorithms [9], they all have a high synchronization delay of $2T$.

In this paper, we present a delay-optimal quorum-based algorithm which reduces the synchronization delay to T, and still has a low message complexity $c * K$, where 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, a site exiting the CS directly sends a *reply* message to the site which enters the CS next. This reduces the 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 [9] 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

A distributed system we consider 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 communicate 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 we use fault tolerant quorum construction algorithm, our algorithm can handle site and communication failures.) 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.

Let U denotes a non-empty 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 [3, 11]:

$$(i)\ (\forall P \in \mathcal{C} :: P \neq \phi \wedge P \subseteq U)$$
$$(ii)\ \textit{Minimality Property} : (\forall P, Q \in \mathcal{C} :: P \not\subset Q)$$
$$(iii)\ \textit{Intersection Property} : (\forall P, Q \in \mathcal{C} :: P \cap Q \neq \phi)$$

For example, $\mathcal{C}$ = {{a,b},{b,c}} is a coterie under U = {a,b,c}, and P = {a,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 has at least one site in common (by the Intersection Property), mutual exclusion is guaranteed. The Minimality Property is not necessary for correctness but is useful for efficiency.

2.2 Basic Idea of Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms [9] associate a request set (quorum) R_i, with a site S_i, such that:

$$(i)\ (\forall i :: S_i \in R_i) \text{ and}$$
$$(ii)\ (\forall i \forall j :: R_i \cap R_j \neq \phi)$$

A 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, a 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 the permission (i.e., *reply* messages) from all the sites in R_i. After having finished the execution of CS, 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 [9] 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 [9] 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 (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, a 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 Figure 1, 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 Figure 1 where mutual exclusion is violated:

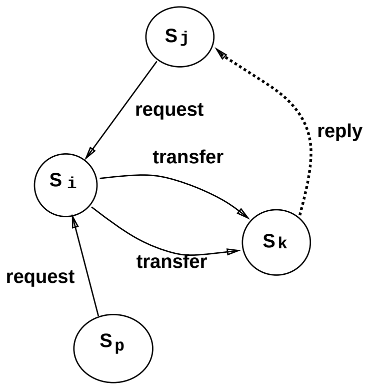

Figure 1: Possible problems

- After S_i has sent a *transfer* message to S_k, S_i cannot send any *reply* to 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 site 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 Figure 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 [7]. 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 massage with smaller sequence number has higher priority.
2. If the massages have equal sequence numbers, the massage with smaller site number has higher priority.

There are seven types of control messages used in our scheme:

request: a $request(sn, i)$ message from a site S_i to a site S_j indicates that S_i with sequence number sn is asking for S_j's permission to enter the CS.

reply: a $reply(i)$ message to a site S_j indicates that S_i grants S_j's request to enter the CS.

release a $release(i, j)$ message to S_k indicates that S_i has exited the CS. If $j \neq max$, S_i has transferred S_k's permission to site S_j.

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

A node S_i maintains the following data structures:

lock: a tuple (sn, j) maintained by each node, 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 $reply(j)$, 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_q: to queue the $inquire(j)$ messages which arrive at S_i earlier than $reply(j)$.

tran_stack: to save all the $transfer$ messages S_i receives. Every entry in this stack is a pair (k, j) which represents a $transfer(k, j)$ message.

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_j, which has already been *locked* by S_i (S_j has sent a *reply* to S_i, but S_j has not received a *yield* or $release$ from S_i), receives a $request$ from S_k. S_j puts S_k's $request$ into its $req\_q(j)$. If S_k's $request$ has the highest priority in $req\_q(j)$, S_j sends a $transfer$ message to S_i, which forwards a *reply* message to S_k after it completes its CS execution. Note that when S_k receives the forwarded *reply*, it gets the permission to enter the CS from S_j even though the *reply* is not directly sent by S_j. S_j may send another $transfer$ message to S_i 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_i only sends a *reply* to the site whose

$request$ is the top entry in $tran\_stack(i)$ and deletes the following entries in $tran\_stack(i)$ 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_i receives a $transfer$ from another site, say S_j, $replied_i[j]$ should be equal to 1; otherwise, the $transfer$ is an outdated $transfer$ and should be discarded.

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

Since there is a danger of deadlock when more than one site simultaneously request 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_i arrives at S_j and S_j has sent a $reply$ to S_k, S_j 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_j 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_q. When a $reply$ arrives, the receiver first checks if there is any $inquire$ that came from the same sender as that of the $reply$. If so, process this $inquire$. If an $inquire$ or $fail$ from a site S_i arrives at S_j after S_j has sent $release$ to S_i, S_j just ignores it.

The following is the formal description of our quorum-based mutual exclusion algorithm.

A: Requesting the Critical Section:

1. /* For a site S_i wishes to enter CS */
   S_i sends $request(sn, i)$ to every site $S_j \in R_i$;
   clear $tran\_stack(i)$, $inq\_q(i)$, and $tran\_set(i)$;
   $failed_i := 0$; $replied_i := 0$; $lock(i) := (max, max)$;
2. Actions when S_j receives a $request(sn, i)$:
   if $lock(j) = (max, max)$
   then $lock(j) := (sn, i)$; send a $reply(j)$ message to S_i;
   else $(sn, k) := lock(j)$; /* Let (sn, k) represent the contents of $lock(j)$ */

case $(req\_q(j) = \phi) \wedge ((sn, i) < lock(j))$:
S_j sends $inquire(j)$ piggybacked with $transfer(i, j)$ to S_k;
case $(req\_q(j) = \phi) \wedge ((sn, i) > lock(j))$:
S_j sends $transfer(i, j)$ to S_k; S_j sends $fail(j)$ to S_i;
case $(req\_q(j) \neq \phi) \wedge ((sn, i) > head(req\_q(j)))$
S_j sends $fail(j)$ to S_i;
case $(req\_q(j) \neq \phi) \wedge ((sn, i) < head(req\_q(j)) < lock(j))$:
S_j sends $fail(j)$ to head $(req\_q(j))$; S_j sends $transfer(i, j)$ to S_k;
case $(req\_q(j) \neq \phi) \wedge ((sn, i) < lock(j) < head(req\_q(j)))$:
S_j sends $inquire(j)$ piggybacked with $transfer(i, j)$ to S_k;
case $(req\_q(j) \neq \phi) \wedge (lock(j) < (sn, i) < head(req\_q(j)))$:
S_j sends $transfer(i, j)$ to S_k;
enqueue $(req\_q(j), (sn, i))$;

3. Actions when a site S_i receives an $inquire(j)$:
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 $yield(i)$ to S_j;
delete all entries sent by S_j in $tran\_stack(i)$;
else enqueue$(inq_q(i), j)$;

4. Actions when a site S_j receives a $yield(k)$:
enqueue $(req\_q(j), lock(j))$;
$(sn, i) :=$ dequeue $(req\_q(j))$; $lock(j) := (sn, i)$; $(sn, p) :=$ head$(req_q(j))$;
send $reply(j)$ piggybacked with $transfer(p, j)$ to S_i;

5. Actions when a site S_i receives a $transfer(k, j)$:
if $reply_i[j] = 1$
then push $(tran\_stack(i), (k, j))$;
else ignore this $transfer$;

6. Actions when a site S_i receives a $reply(j)$:
$replied_i[j] := 1$;
if $j \in inq\_q(i)$
then delete j from $inq\_q(i)$; Execute A.3 as if S_i receives $inquire(j)$;

7. Actions when a site S_i receives a $fail(j)$:
$failed_i := 1$;
for any $j \in inq\_q(i)$
delete j from $inq\_q(i)$; Execute A.3 as if S_i receives $inquire(j)$;

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:
while $tran\_stack(i) \neq \phi$
$(j, k) := pop(tran\_stack(i))$;
S_i sends $reply(k)$ to S_j; $tran\_set(i) := tran\_set(i) \cup (j, k)$;
delete other entries sent by S_k in $tran\_stack(i)$;

For each $S_k \in R_i$:

  if $\exists (j,k) \in tran\_set(i)$: /* there exists an entry sent by S_k in $tran\_set(i)$ */

  then send $release(i,j)$ to S_k;

  else send $release(i,max)$ to S_k;

2. Actions when a site S_k receives a $release(i,j)$:

   if $j \neq max$

   then $lock(k) := (sn,j)$; delete (sn,j) from $req\_q(k)$;

     if $req\_q(k) \neq \phi$

     then $(sn,p) :=$ head$(req_q(k))$

       if $(sn,p) < (sn,j)$

       then send $inquire(k)$ piggybacked with $transfer(p,k)$ to S_j;

       else send $transfer(p,k)$ to S_j;

   else if $req\_q(k) = \phi$

     then $lock(k) := (max,max)$;

     else $(sn,p) :=$ dequeue$(req_q(k))$; $lock(k) := (sn,p)$;

       if $req\_q(k) = \phi$

       then send $reply(k)$ to S_p;

       else $(sn,q) :=$ head$(req_q(k))$;

         send $reply(k)$ piggybacked with $transfer(q,k)$ to S_p.

4 Correctness Proof

Theorem 1 *Mutual exclusion is achieved.*

Proof. Assume the contrary that two sites S_i and S_j are executing the CS simultaneously. From the Coterie Intersection Property: $(\forall P, Q \in \mathcal{C} :: P \cap Q \neq \phi)$, we know that S_i's quorum R_i and S_j's quorum R_j have at least one common site, say S_{ij}. From step B of the algorithm, if S_i and S_j are executing the CS simultaneously, both of them must have locked S_{ij}'s *reply* at the same time. In our algorithm, before a site enters the CS, it may receive a *reply* message from the arbiter site, yield the *reply*, and receive a *reply* from the arbiter site again. In the following proof, we only concern about the last *reply* that a site receives from any site in its quorum before it enters the CS; that is, both S_i and S_j must have locked S_{ij}'s *reply* messages and do not release them before they enter the CS. We prove that this is impossible.

Case 1: Both S_i and S_j obtain *reply* messages from S_{ij} directly (without the transfer of some other site). Without loss of generality, assume S_{ij} sends a *reply* to S_j after it has sent a *reply* to S_i. From the algorithm, after S_{ij} has sent a *reply* to S_i, $lock = (sn, i)$. Without yielding to others, S_i does not release the *reply* until it gets out of the CS (release can only happen in step C); that is, $lock \neq (max, max)$ until S_i gets out of the CS. Therefore, S_j cannot get a *reply* directly from S_{ij} before S_i gets out of the CS.

Case 2: S_i obtains the *reply* from S_{ij} directly, while S_j gets the *reply* indirectly (by the transfer of some other site). There are two possible situations:

Case 2.1: S_i gets a *reply* directly from S_{ij} after S_{ij} sends a *reply* to a site, say S_k. In this case, S_{ij} is locked by S_k and $S_j \in tran\_stack(k)$. From the algorithm, a site can only transfer a *reply* in C.1. In C.1, after sending a *reply* on behalf of S_{ij}, S_k also sends a *release* which asks S_{ij} to change its *lock* to be (sn, j) according to C.2. Then, S_j is locking S_{ij}'s *reply*. Since S_i can only directly obtain S_{ij}'s *reply*, from the result of Case 1, S_i cannot get S_{ij}'s *reply* until S_j releases its lock on S_{ij}'s *reply*.

Case 2.2: S_i gets the *reply* directly from S_{ij} before S_{ij} sends a *reply* to any other site. Then, S_i is locking S_{ij}'s *reply*. There are two possibilities for S_j to get a *reply* indirectly from S_{ij}. First, S_j gets the *reply* indirectly from S_{ij} by S_i ($S_j \in tran\_stack(i)$). From step C, a site can only transfer a *reply* on behalf of other site when it gets out of the CS. Therefore, S_j can only get the *reply* indirectly after S_i releases the CS. Secondly, S_j gets the *reply* indirectly from S_{ij} by S_{k_0}, S_{k_0} gets the *reply* from S_{ij} by S_{k_1}, $\cdots$, and $S_{k_{m-1}}$ gets the *reply* from S_{ij} by S_{k_m}. In the end, S_{k_m} gets the *reply* from S_{ij} directly. S_i must get the *reply* directly from S_{ij} before S_{K_0} gets the *reply*; otherwise, there is a contradiction from Case 2.1. Similarly, S_i gets the *reply* before $S_{K_1}, \cdots, S_{k_{m-1}}$. Based on Case 1, it is impossible for S_i and S_{k_m} to obtain *reply* messages directly from S_{ij} at the same time.

Case 3: Both S_i and S_j obtain *reply* messages from S_{ij} indirectly. At the beginning, no site is in the CS and $lock = (max, max)$. Thus, a site can only get S_{ij}'s *reply* directly, and later by the transfer of other sites. Based on Case 1 and Case 2, before S_{ij} asks the site which is locking S_{ij}'s *reply* to transfer *reply* messages to more than one site, there is only one site locking S_{ij}'s *reply*. Suppose a site, say S_k, locks S_{ij}'s *reply*. The only possibility of Case 3 is that S_{ij} asks S_k to transfer *reply* messages to both S_i and S_j. According to C.1, when S_k exits its CS, it responses to only one $transfer$ from each site in its quorum. Therefore, S_k cannot send two *reply* messages to S_i and S_j. 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.

Case 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.

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

Case 1: $(req\_q(j) = \phi) \wedge ((sn, i) > lock(j))$. 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: $(req\_q(j) = \phi) \wedge ((sn, i) < lock(j)) OR (req\_q(j) \neq \phi) \wedge ((sn, i) < lock(j) < head(req\_q(j)))$. There are two cases depending on whether the inquired site has replied a *yield* or not.

Case 2.1 (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.

Case 2.2 (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: $(req\_q(j) \neq \phi) \wedge ((sn, i) > head(req\_q(j)))$. 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: $(req\_q(j) \neq \phi) \wedge ((sn, i) < head(req\_q(j)) < lock(j))$. There are two cases depending on whether the inquired site has replied a *yield* or not.

Case 4.1 (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.

Case 4.2 (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: $(req\_q(j) \neq \phi) \wedge (lock(j) < (sn, i) < head(req\_q(j)))$. 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 upon the inter-request 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 a 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 *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.

Case 2.1: S_i is locking S_k's reply. In this case, S_k sends a $transfer(j, k)$ 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.

Case 2.2: $S_p (p \neq i)$ 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 transferred *reply* from S_i; that is, when S_j receives a *reply* or 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[1], a site needs to wait for a long time to enter the CS. During the long wait, its deadlock resolution is more likely to have been finished before it receives a *reply* or transferred *reply* from the last site executing the CS. Thus, based on Theorem 4, the synchronization delay is T.

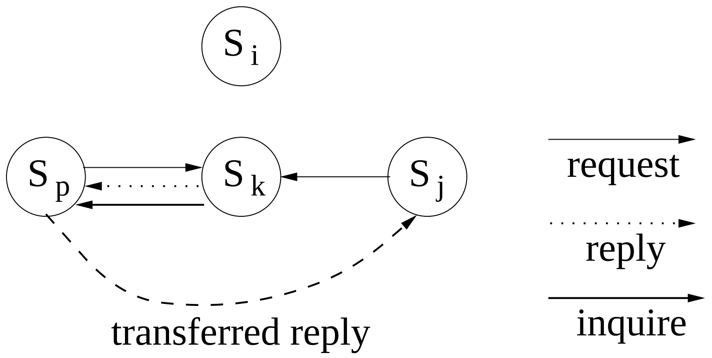

Figure 2: Futher reduce synchronization delay

Note that in the proof of Theorem 4, if S_j's deadlock resolution has not been finished when it receives a *reply* or transferred *reply* from S_i, the synchronization delay may be longer than T. This can be explained by the following example. In Figure 2, S_i is excuting 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

[1] "Under heavy load" does not necessary mean that the whole system is under heavy load. It may occur when only one arbiter site is under heavy load.

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 $2T$.

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 *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 [7], Ricart-Agrawal algorithm [16], etc., needs $2T$. Hence, we consider this special case as 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 satify the requirement of Theorem 4. For example, in Figure 2, S_p may have not gotten a *yield* when it receives the *inquire* from S_k, since the the message delays may be different and the $2T$ 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 [7], Ricart-Agrawal algorithm [16], etc., even though the synchronization delay is T, the T is the longest message delay in the system. However, the T in our algorithm may be significantly shorter than the longest message delay. For example, when using fault-tolerant quorums [2, 14], our algorithm may not need to wait for those long delayed messages.

According to [18], 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 $E < T$

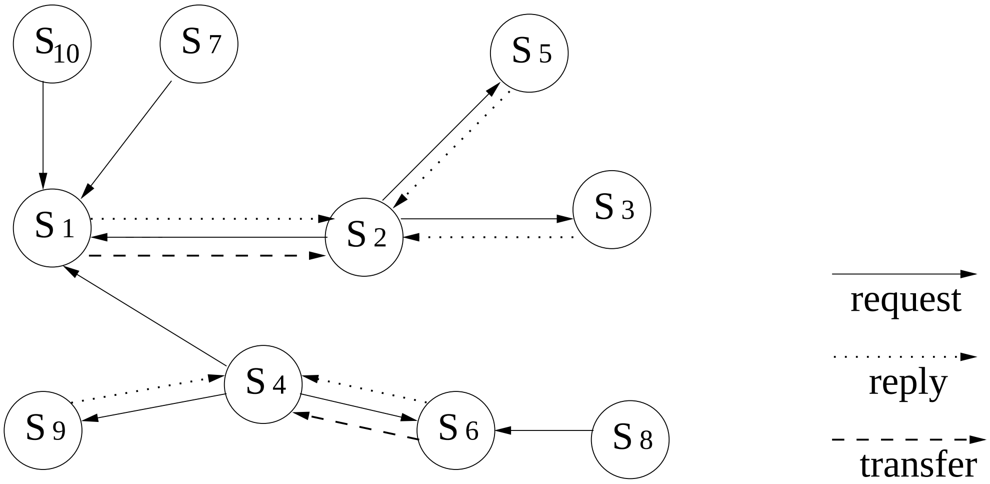

Figure 3: Synchronization delay when $E < T$

When $E < T$, the synchronization delay may be longer than T. This can be explained by the following example. In Figure 3, 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 any $transfer$ 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 $transfer(7, 1)$ 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 $2T$[2], 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 denotes the probability of S_j in S_k's quorum. The synchronization delay $\mathcal{D}$ is given by the following expression:

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

If we use Maekawa's quorum construction algorithm [9], 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 Figure 3, when S_1 receives S_4's *request*, it sends $transfer(4, 1)$ to S_2. When S_1 receives S_7's *request*, since it has sent a $transfer$ to S_2, it sends $transfer(7, 1)$ 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 any $transfer$ until it receives a *release* from S_2. When S_1 receives the *release*, it sends $transfer(10, 1)$ to S_7. In summary, the arbiter site sends $transfer$ messages one step ahead.

[2]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.

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

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 [9], 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, 2, 5, 6, 8, 9, 11, 12, 13, 14] 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 [21] 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. We consider four well known fault-tolerant quorum construction algorithms.

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	T	N	N
Singhal's heuristic	T	$N/2$	N
Raymond	$T(\log N)/2$	$\log N$	4

Table 1: **A comparison of performance (ll = light load, hl = heavy load)**

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. The quorum size in the tree algorithm is $\log N$ in the best case and becomes $\frac{N+1}{2}$ in the worst case.

In the HQC or Hierarchical Voting Consensus algorithm [5], 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 [2] 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 message overhead. The quorum size is $\frac{\frac{N}{G}+1}{2}\sqrt{G}$, where G is the group size.

The Rangarajan-Setia-Tripathi algorithm [14] in some sense is a dual of the Grid-set algorithm [2]. Specifically, they use majority voting at the lower (subgroup) level and a Maekawa-like grid structure at the higher level. With this change, the quorum size in this algorithm reduces to $\frac{G+1}{2}\sqrt{\frac{N}{G}}$, where G is the subgroup size.

If our algorithm uses the fault tolerant quorum constructed by any of these algorithms [1, 2, 5, 14], 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 [14] (or the Grid-set[2]) and the tree algorithm [1] (or the HQC algorithm [5]). 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 $request(sn, i)$ is in its $req\_q(j)$, $tran\_stack(j)$ or $lock(j)$:

   Case 1: $(sn, i) \in req\_q(j)$: If (sn, i) is the top entry in $req\_q(j)$ and $req\_q(j)$ has more than one entry, S_j deletes (sn, i) from $req\_q(j)$ and sends $transfer(tail(head(req\_q(j))), j)$ to the site in $lock(j)$. Otherwise, S_j just deletes (sn, i) from $req\_q(j)$.

   Case 2: $(sn, i) \in tran\_stack(j)$: Delete (sn, i) from $tran\_stack(j)$;

   Case 3: $(sn, i) \in lock_j$: 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 $req\_q(j) = \phi$
   then $lock(j) := (max, max)$;
   else $(sn, p) :=$ dequeue$(req_q(j))$; $lock(j) := (sn, p)$;
       if $req\_q(j) = \phi$
       then send $reply(j)$ to S_p;
       else $(sn, q) :=$ head$(req_q(j))$;
           send $reply(j)$ piggybacked with $transfer(q, j)$ to S_p.

7 Conclusions

Quorum is an attractive approach to provide mutual exclusion in distributed systems since it has low massage complexity and high resiliency. After the first quorum-based algorithm [9] was proposed by Maekawa more than a decade ago, many algorithms [1, 2, 5, 6, 8, 11, 12, 13, 14] have been proposed to construct different quorums to reduce the message complexity or increase the resiliency to site and communication failures. However, not much work has been done towards minimizing the synchronization delay. Because all existing quorum-based algorithms depend on Maekawa's algorithm to ensure mutual exclusion, they all have high synchronization delay ($2T$).

In this paper, we presented 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 $\log N$). In our algorithm, 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, a site exiting the CS directly sends a *reply* message to the site to enter the CS next. Thus, after one site exits the CS, it only takes one message delay before the next site enters the CS, which reduces the synchronization delay from $2T$ in Maekawa's algorithm to T. 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.

References

[1] D. Agrawal and A.E. Abbadi. "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion". *ACM Transactions on Computer Systems*, February 1991.

[2] S.Y. Cheung, M.H. Ammar, and M. Ahamad. "The Grid Protocol: A high performance scheme for maintaining Replicated data". *IEEE Trans. knowl. Data Eng.*, June 1992.

[3] H. Garcia and D. Barbara. "How to assign Votes in a Distributed System". *J. ACM*, May 1985.

[4] J. Helary, A. Mostefaoui, and M. Raynal. "A General Scheme for Token- and Tree-based Distributed Mutual Exclusion Algorithms". *IEEE Transactions on Parallel and Distributed Systems*, 5:1185–1196, Nov. 1994.

[5] A. Kumar. "Hierarchical Quorum Consensus: A new Algorithm for managing Replicated Data". *IEEE Trans. Computers*, pages 996–1004, September 1991.

[6] Yu-Chen Kuo and Shing-Tsaan Huang. "A Geometric Approach for Constructing Coteries and k-Coteries". *IEEE Transactions on Parallel and Distributed Systems*, 8:402–411, April 1997.

[7] L. Lamport. "Time, Clocks and Ordering of Events in Distributed Systems". *Comm. of the ACM*, July 1978.

[8] Wai-Shing Luk and Tien-Tsin Wong. "Two New Quorum Based Algorithms for Distributed Mutual Exclusion". *Proceedings of the* 17^{th} *International conference on Distributed Computing Systems*, May 1997.

[9] M. Maekawa. "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems". *ACM Transactions on Computer Systems*, May 1985.

[10] M. Naimi and M. Trehel. "An Improvement of the Log(n) Distributed Algorithm for Mutual Exclusion". *Proceedings of the 7th International Conference on Distributed Computing System*, pages 371–375, 1987.

[11] M.L. Neilsen. "Fault-Tolerant Distributed Mutual Exclusion". *Seventh International Conference on Parallel and Distributed Computing Systems*, pages 539–542, 1994.

[12] W.K. Ng and C.V. Ravishankar. "Coterie Templates: A New Quorum Construction Method". *Proceedings of the* 15^{th} *International conference on Distributed Computing Systems*, pages 92–99, May 1995.

[13] D. Peleg and A. Wool. "Crumbling Walls: A Class of Practical and Efficient Quorum Systems". *Proceedings of* 14^{th} *ACM Symposium on Principles of Distributed Computing*, pages 120–129, August 1995.

[14] S. Rangarajan, S. Setia, and S.K. Tripathi. "A Fault-Tolerant Algorithm for Replicated Data Management". *IEEE Transactions on Parallel and Distributed Systems*, pages 1271–1282, December 1995.

[15] K. Raymond. "A Tree-based Algorithm for Distributed Mutual Exclusion". *ACM Transactions on Computing systems*, pages 61–77, February 1989.

[16] G. Ricart and A.K. Agrawal. "An Optimal Algorithm for mutual Exclusion in Computer Networks". *Communication of the ACM*, January 1981.

[17] M. Singhal. "A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed System". *IEEE Transactions on Computers*, May 1989.

[18] M. Singhal. "A Class of Deadlock-Free Maekawa-type Algorithms for Mutual Exclusion in Distributed Systems". *Distributed Computing*, 4:131–138, February 1991.

[19] M. Singhal. "A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems". *IEEE Transactions on Parallel and Distributed Systems*, January 1992.

[20] M. Singhal. "A Taxonomy of Distributed Mutual Exclusion". *Journal of Parallel and Distributed Computing*, 18:94–101, May 1993.

[21] T.H. Thomas. "A majority consensus approach to concurrency control for multiple copy databases". *ACM Trans. Database Systems*, pages 180–209, June 1979.