A Delay-Optimal Quorum-Based Mutual Exclusion Scheme with Fault-Tolerance Capability *

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

Yi Deng, Naphtali Rishe, and Wei Sun
School of Computer Science
Florida International University
Miami, FL 33199
{deng,rishe,weisun}@fiu.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 tradeoff in mutual exclusion algorithms. Lamport's algorithm and Ricart-Agrawal algorithm both have a synchronization delay of $T$, 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 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 the low message complexity $O(K)$ ($K$ is the size of the quorum, which can be as low as $\log N$). A correctness proof and detailed performance analysis are provided.

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

1 Introduction
In distributed systems, many applications involving replicated data, atomic commitment, distributed shared memory, and others require that a resource be 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 can not get the shared resource must wait, the performance of the mutual exclusion algorithm 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 [17] 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 [6] to implement distributed mutual exclusion. For each CS execution, each site needs to get permissions 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 [13] is an optimization of Lamport's algorithm that reduces the release message by cleverly merging them with reply messages. This merging is achieved by deferring the lower priority request. 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 [16] 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 [8], a set of sites called a quorum is associated with each site, and this set has a non-empty intersection with the sets corresponding to every other site. To execute CS, a site only locks all sites in its quorum; thus, message complexity is
dramatically reduced. At light load, a site needs to exchange $3 \cdot (\sqrt{N} - 1)$ messages to achieve mutual exclusion. At heavy load, because of the need to handle deadlocks, the message complexity becomes $5 \cdot (\sqrt{N} - 1)$. However, the synchronization delay becomes $2T$ as opposed to $T$ in other algorithms. This is 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 [15] 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 [14], 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 [9, 12] on the average require only $O(\log N)$ messages to execute the critical section; however, the average delay in these algorithms is also $O(\log N)$. The worst case delay of the algorithm in [9] 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. Besides the long delay, token-based algorithms suffer from token loss problem [1].

Recently, quorum-based mutual exclusion algorithms, which are a generalization of Maekawa's algorithm, have attracted considerable attention. Many algorithms [1, 2, 4, 5, 7, 8, 10, 11] 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 [8], they all have a high synchronization delay ($2T$).

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 $c \cdot K$, 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 will enter the CS. 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 quorum being used. $K$ is $\sqrt{N}$ if we use Maekawa's quorum construction algorithm [8] and $K$ becomes $\log N$ when we use Agrawal-Abadi 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 describes the system model. 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 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 $C$ is a set of sets, where each set $g$ in $C$ is called a quorum. The following conditions hold for quorums in a coterie $C$ under $U$ [3]:

1. $(\forall g \in C)[g \neq \emptyset \land g \subseteq U]$;
2. Minimality Property : $(\forall g, h \in C)[g \subseteq h]$; and
3. Intersection Property : $(\forall g, h \in C)[g \cap h \neq \emptyset]$.

For example, $C=\{\{a,b\}, \{b,c\}\}$ is a coterie under $U=\{a,b,c\}$, and $g=\{a,b\}$ 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 resource in the network, a site, say $S_i$, is required to receive permissions from a quorum of $S_i$ in the system. If all sites in the quorum of $S_i$ grant permissions to $S_i$, $S_i$ is allowed to access the 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.
3 A Delay-Optimal Quorum Based Algorithm

Our algorithm reduces the synchronization delay to $T$ as follows: When a site exists the CS, 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 directly sends a reply message to the site to enter the CS next. Although the idea may sound simple, its implementation is difficult in order to ensure mutual exclusion and to avoid deadlocks. For example, there are two ways for a site $S_i$ to get permission to enter the CS from a site $S_j$: First is to get the permission from $S_j$ directly; the other is to get the permission from a site $S_k$ which has gotten the permission from $S_j$ and works as the proxy of $S_j$. Then, after $S_i$ exists the CS, if $S_i$ has sent a reply to $S_j$ on behalf of $S_j$, $S_j$ cannot send reply to any other site to ensure mutual exclusion. If $S_k$ has not sent a reply to any site on behalf of $S_j$, $S_j$ should send a reply to $S_j$ to avoid deadlock. Also, to deal with out-of-order request messages, 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 $S_i$ may come from different channels: from $S_i$ or $S_i$'s proxy. 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 the mutual exclusion. There are many other issues that must be dealt with. Before presenting the algorithm, we first introduce control messages and data structures used in our algorithm.

3.1 Control Messages and Data Structures

Every site $S_i$ has a req-set($i$) which is determined by the quorum algorithm. In order to enter the CS, each site must get permissions from all the sites in req-set($i$).

Every request message is assigned a timestamp (the sequence number and the site number) according to Lamport’s scheme [6]. 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 smaller site number has higher priority.

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

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

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

release A release($i$) message to $S_i$ indicates that $S_i$ has exited the CS. If $j \neq \text{max}$, $S_i$ has transferred $S_i$'s permission to a site $S_j$ which is in $S_i$'s tran_stack (defined later).

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 req-set($j$).

fail: A fail($i$) message from $S_i$ to $S_j$ indicates that $S_i$ can not grant $S_j$'s request because it has currently granted 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 is waiting for $S_j$'s permission to enter the CS.

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

A node $S_i$ maintains the following data structures:

lock: A tuple $(s_n, j)$ maintained by each node, where $j$ is the site number of the request site to which $S_i$ has granted a reply, and $s_n$ is the sequence number of the request message. lock is initialized to $(\text{max}, \text{max})$, 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 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[$j$] to 1.

req_queue: To queue the received request messages. Each entry in this queue is a tuple $(s_n, j)$ which is the timestamp of a request. The req_queue is a priority queue (the request with the highest priority is on the top of the queue).

inq_queue: 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.

The algorithm does not depend on any particular quorum construction method and works for any types of quorums.

3.2 The Algorithm

To enter the CS, a site $S_i$ requests permission from each site in its quorum. If $S_i$ has gotten permissions from all members in its quorum, it can enter the CS; otherwise, it waits for permission from those sites.

When a site $S_i$ (or $S_j$) has permission, it sends a request to all other sites in the quorum. If a site $S_k$ has not received a reply within a certain time, it sends a request to all other sites in the quorum. This process is repeated until all sites in the quorum have granted permission.

When a site $S_i$ receives a request from another site, it checks whether its request message is sent from another site. If its request message is sent from an already sent request message, it sends a replied message to the site that sent the request message. If its request message is sent from a new request message, it sends a request message to all other sites in the quorum.

The algorithm ensures mutual exclusion and prevents deadlocks by using the priority scheme and the FIFO assumption.
otherwise, it continues to wait for the permission of all those sites.

When a site $S_j$, which has already been locked by $S_i$ ($S_j$ has sent a reply to $S_i$), receives a request from $S_k$. $S_j$ puts $S_k$’s request in its req-queue($j$). If $S_k$’s request has the highest priority in req-queue($j$), $S_j$ sends a transfer message to $S_k$, which forwards a reply message to $S_k$ after it completes its CS execution. Note that when $S_k$ receives the forwarded reply message, 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 several transfer messages to $S_k$ in response to out-of-order request messages. Upon exiting the CS, site $S_i$ only sends 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 $S_j$ receives a transfer from another site, say $S_j$, replied[$j$] should be equal to $1$; otherwise, the transfer is an outdated transfer and should be discarded.

When a site $S_j$ receives a release message 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 message. 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-queue($j$) is not empty, $S_j$ sends a transfer to $S_k$ based on the top entry in req-queue($j$). $S_j$ sends a reply to the top entry site in req-queue($j$) if $S_i$ has not transferred the 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 high priority from $S_i$ arrives at $S_j$, such that $S_j$ 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 message. Otherwise, it returns a release after it completes its CS execution. We use piggybacking to reduce message complexity. For example, whenever a site sends an inquire in response to a high 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-queue. When a reply arrives, the algorithm first checks to see if there are 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 delay-optimal 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 $S_j \in$ req-set($i$);
   clear tran_stack($i$), inq_queue($i$), 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_queue($j$) = $\phi$) \land ((sn,$i$) < lock($j$)):
   $S_j$ sends inquire($j$) and transfer($i,j$) to $S_k$;
   case (req_queue($j$) = $\phi$) \land ((sn,$i$) > lock($j$)):
   $S_j$ sends transfer($i,j$) to $S_k$;
   case (req_queue($j$) \neq $\phi$) \land
   ((sn,$i$) > head(req_queue($j$))):
   $S_j$ sends$\neq$ fail($j$) to $S_i$;
   case (req_queue($j$) \neq $\phi$) \land
   ((sn,$i$) < head(req_queue($j$)) < lock($j$)):
   $S_j$ sends$\neq$ fail($j$) to head (req_queue($j$));
   $S_j$ sends transfer($i,j$) to $S_k$;
   case (req_queue($j$) \neq $\phi$) \land
   ((sn,$i$) < head(req_queue($j$)):
   $S_j$ sends inquire($j$) and transfer($i,j$) to $S_k$;
   case (req_queue($j$) \neq $\phi$) \land
   (lock($j$) < (sn,$i$) < head(req_queue($j$))):
   $S_j$ sends transfer($i,j$) to $S_k$;
   enqueue (req_queue($j$), (sn,$i$));

3. Actions when a site $S_j$ receives an inquire($j$):
   if (replied,$j$ = 1) \land (failed,$i$ = 1)
   /* $S_j$ has received a fail or sent a yield */
   then replied,$j := 0$; failed,$i := 1$;
   send a yield($i$) to $S_j$;
   delete all entries sent by $S_j$ in tran_stack($i$);
   else enqueue (inq_queue($i$), $j$);

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

5. Actions when a site $S_j$ receives a transfer($k,j$):
   if reply,$j = 1$
   then push (tran_stack($i$), (k,$j$));
   else ignore this transfer;

6. Actions when a site $S_j$ receives a reply($j$):
   replied,$j := 1$;
   if $j \in$ req_queue($i$)
   then delete $j$ from req_queue($i$);
   Execute A.3 as if $S_i$ receives inquire($j$);

7. Actions when a site $S_j$ receives a fail($j$):
   failed,$i := 1$;
   for any $j \in$ req_queue($i$)
   delete $j$ from req_queue($i$);
   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
 req_set($i$), replied,$[k] = 1$.

C: Releasing the Critical Section:
Actions when $S_i$ exits the CS:

\[ \begin{align*}
\text{while } & \text{trans\_stack}(i) \neq \phi \\
& (j, k) := \text{pop}(\text{trans\_stack}(i)); \\
& S_i \text{ sends } \text{reply}(k) \text{ to } S_j; \\
& \text{trans\_stack}(i) = \text{trans\_stack}(i) \cup (j, k); \\
& \text{delete other entries sent by } S_k \text{ in } \text{trans\_stack}(i); \\
\end{align*} \]

For each $S_k \in \text{req\_set}(i)$:

\[ \begin{align*}
& \exists (j, k) \in \text{trans\_set}(i); \\
& \text{/* there exists an entry sent by } S_k \text{ in } \text{trans\_set}(i) */ \\
& \text{then send release}(i, j) \text{ to } S_k; \\
& \text{else send release}(i, \text{max}) \text{ to } S_k; \\
\end{align*} \]

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:

\[ V G \cap H \neq \phi. \]

We know that the quorums ($\text{req\_set}$) of $S_i$ and $S_j$ at least have one common site, say $S_k$. From step B of the algorithm, if $S_i$ and $S_j$ are executing the CS simultaneously, both of them must have locked $S_k$'s reply at the same time. We prove that this is impossible.

**Case 1:** Both $S_i$ and $S_j$ obtain reply messages from $S_k$ directly (without the transfer of another site). Assume $S_i$ sends a reply to $S_j$ after it has sent a reply to $S_k$. From the algorithm, after $S_i$ has sent a reply to $S_k$, the lock is changed to $(sn, i)$. There are two possible situations:

**Case 1.1:** $S_i$ does not send a yield to $S_k$ after it gets the reply. In this case, $S_i$ will not release the reply until it gets out of the CS (release can only happen in step C), which means that the lock is not equal to $(\text{max, max})$ until $S_i$ gets out of the CS. Therefore, $S_j$ cannot get a reply directly from $S_k$ before $S_i$ gets out of the CS.

**Case 1.2:** $S_j$ sends a yield to $S_k$. According to A.3, $S_i$ sends yield to $S_j$ only when it is locking $S_i$'s reply. After sending the yield, $S_i$ assumes it has not received the reply from $S_i$ and releases the lock. As a result, when $S_i$ obtains a reply from $S_i$, $S_i$ is not locking $S_i$'s reply.

**Case 2:** Site $S_i$ obtains the reply from $S_i$ directly, while $S_j$ gets the reply indirectly (by the transfer of another site). There are two possible situations:

**Case 2.1:** $S_i$ gets a reply directly from $S_k$ before $S_k$ sends reply to any other site, then $S_i$ is locking $S_i$'s reply. In order to get a reply indirectly from $S_i$, $S_j$ can only be in the $\text{trans\_stack}(i)$ or in a site, say $S_k$'s $\text{trans\_stack}(k)$. 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 reply indirectly after $S_i$ releases the CS.

**Case 2.2:** $S_i$ gets a reply directly from $S_k$ after $S_i$ sends reply to a site, say $S_j$. In this situation, $S_k$ is locked by $S_k$, and sends transfer to $S_k$, then $S_j$ is in $\text{trans\_stack}(k)$. From the algorithm, a site can only transfer a reply in C.1. In C.1, after sending a reply on behalf of $S_i$, $S_k$ also sends a release which asks $S_i$ to change its lock to be $(sn, j)$ according to C.2. Then, $S_j$ is locking $S_i$'s reply. Since $S_i$ can only directly obtain $S_i$'s reply, from the result of Case 1, it can get $S_i$'s reply only until $S_j$ releases its lock on $S_i$'s reply.

**Case 3:** Both $S_i$ and $S_j$ obtain reply messages from $S_k$ indirectly. When our algorithm starts, a site can only get $S_k$'s reply directly, and later by the transfer of other sites. Based on Case 1 and Case 2, before $S_i$ asks the site which is locking $S_i$'s reply to transfer a reply to more than one site, there is only one site locking $S_i$'s reply. Suppose a site, say $S_k$, locks $S_i$'s reply. Then, the only possibility of Case 3 is that $S_i$ asks $S_k$ to transfer a reply to both $S_i$ and $S_j$. According to C.1, when $S_k$ exits its CS, it responses to at most one $\text{trans\_stack}$ from any sender. Therefore $S_k$ can not 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 be able to execute the CS because each 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 site in the cycle.

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

**Case 1:** Site $S_k$ sends a yield to $S_j$. Then, $S_j$ sends a reply to $S_j$ according to A.3 and A.4, and the cycle is broken.

**Case 2:** Site $S_j$ 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, $S_p$ must have lower priority than $S_k$. Otherwise, $S_k$ gets a failure.

A.3. For the reply of a low priority site, CS or sends yield to continue to other sites which enters the CS for its reply and another site.

**Theorem 3**

**Proof.** Starvation scenario to enter its CS is repeatedly entered by a site entering a site, so $S_i$ must have a site, say $S_k$, which has requested, and the destination is reliable. In order to be assigned a sequence number, a message will have the sequence number of each site in the request sent to it. If a site $S_i$ does not receive a reply, a contradiction occurs.

5 A Performance

The performance is studied under a light load at the message pig, as one message protocol relative to the message cost of other communication cost.

5.1 Per

Suppose the loads, the contentions, the CS requires release me or the CS execution time. The system semantics are as the arrival time (E) is the time spent by any mutual.

5.2 Per

Suppose a message $S_j$ has sent
Theorem 3 Starvation is impossible.

Proof. Starvation occurs when a site waits indefinitely to enter its critical section while other sites are repeatedly entering and exiting their CS. Assume there is a starving site $S_j$. From Theorem 2, there is always a site entering and exiting the CS. The starving site $S_j$ must have sent request messages to all the sites in $req_set(i)$, 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, $S_j$'s request will have the highest priority among all the request messages received by each site in $req_set(i)$. Then, each site in $req_set(i)$ has sent a reply to $S_j$, or has asked other site to transfer a reply to $S_j$. Therefore, $S_j$ receives all the replies 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 will not increase the communication cost significantly.

5.1 Performance Under Low Load

Suppose the average quorum size is $K$. Under light loads, 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.

The synchronization delay in light load becomes meaningless because it depends upon the inter-request arrival time. The response time in light load is $2T + E$ (E is the CS execution time) which is necessary for any mutual exclusion algorithms in light traffic load.

5.2 Performance Under Heavy Load

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

Case 1: $(req_queue(j) = \phi) \land ((sn, i) > lock(j))$: 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: $(req_queue(j) = \phi) \land ((sn, i) < lock(j))$ OR $(req_queue(j) \neq \phi) \land ((sn, i) < lock(j) < head(req_queue(j)))$: There are two cases depending on whether the inquired site has replied 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, $(K-1) transfer$ 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: $(req_queue(j) \neq \phi) \land ((sn, i) > head(req_queue(j)))$: 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: $(req_queue(j) \neq \phi) \land ((sn, i) < head(req_queue(j)) < lock(j))$: There are two cases to consider 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 $(K-1) request$, $(K-1) fail$, $(K-1) transfer$, $(K-1) reply$ and $(K-1) release$ 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: $(req_queue(j) \neq \phi) \land (lock(j) < (sn, i) < head(req_queue(j)))$: The execution of a CS requires $(K-1) request$, $(K-1) fail$, $(K-1) transfer$, $(K-1) release$, $(K-1) reply$, 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.

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, the 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 needs one message delay for the next
site to obtain the reply message from the site locking the arbiter site. Under heavy load, a site that is waiting to excute the CS has enough time to obtain all reply messages except the reply from the site in the CS before the site in the CS exists the CS. Thus, the synchronization delay is mainly determined by the site in the CS (not other sites). Therefore, our algorithm reduces the synchronization delay from $2T$ in Maekawa's algorithm to $T$. This has two very beneficial implications: First, at heavy loads, the rate of CS execution (i.e., throughput) is doubled. Second, at heavy loads, the waiting time of requests is nearly reduced to half because the CS executions proceed with twice the rate.

Since the site that exists 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-optimial quorum-based mutual exclusion algorithm.

### 5.3 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 [8], and $K = \log N$ when we use 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, 4, 5, 7, 8, 10, 11] 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 [18] 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.

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 $N^{0.5}$ in the worst case.

In HQC or Hierarchical Voting Consensus [4], 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 $N^{0.43}$.

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 $\frac{N+1}{2}\sqrt{G}$, where $G$ is the group size.

The Rangarajan-Setia-Tripathi algorithm [11] 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, 4, 11], 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 Rangarajan-Setia-Tripathi algorithm [11] (or the Grid-set [2]) and the tree algorithm [1] (or HQC algorithm [4]). 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 $S_j \in req.set(i)$. If so, makes $S_j$ inaccessible, releases all the resources it has gotten, and executes the quorum construction algorithm to select another quorum.

2. $S_j$ checks whether $req.queue(i)$ is not empty.

Case 1: $S_j$ is the only entry in the queue and sends the site from $req.queue(i)$ to the CS.

Case 2: $S_j$ is not the only entry in the queue and sends the site from $req.queue(i)$ to the remaining sites.

Case 3: $S_j$ is not the only entry in the queue and sends the site from $req.queue(i)$ to the remaining sites.

### 7 Conclusion

Quorum mutual exclusion algorithms first proposed by Maekawa [1, 2, 4, 5] differ in their definition or implementation of the quorum and fail to cause all the sites in a group to enter the CS.

In the exclusion delay to $O(K)$ (low as long as being a root of the CS) message one site delay before the synchronization of quorum, the algorithmic mutual can be used quorum.
2. $S_j$ checks whether $S_i$'s request $(sn, i)$ is in its req_queue(j), tran_stack(j) or lock(j):
   Case 1: $(sn, i) \in req\_queue(j)$: If $(sn, i)$ is the top entry in req_queue(j) and req_queue(j) has more than one entry, $S_j$ deletes $(sn, i)$ from req_queue(j) and sends transfer\_head\_call\_reply\_queue(j), j) to the site in lock(j). Otherwise, $S_j$ just deletes $(sn, i)$ from req_queue(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, $S_j$ is locking $S_i$. Therefore, $S_j$ releases itself from $S_i$, and sends reply\_pigg\_back\_protocol\_call\_reply\_queue(j) to the site whose request is the top entry in req_queue(j). The formal description is as follows:
   if req_queue(j) == \emptyset
   then lock(j) := (max, max);
   else (sn, p) := dequeue(req_queue(j));
   lock(j) := (sn, p);
   if req_queue(j) == \emptyset
   then send reply\_queue\_protocol\_call\_reply\_queue(j) to $S_p$;
   else (sn, q) := head(req_queue(j));
   send reply\_queue\_protocol\_call\_reply\_queue(j) and transfer\_queue\_protocol\_call\_reply\_queue(j) to $S_p$.

7 Conclusions

Quorum is an attractive approach to provide mutual exclusion in distributed systems since it has low message complexity and high resiliency. After the first quorum-based algorithm [8] was proposed by Maekawa more than a decade ago, many algorithms [1, 2, 4, 5, 7, 10, 11] have been proposed to construct different quorums, which 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 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 quorum 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