A Quorum-Based Group Mutual Exclusion Algorithm for a Distributed System with Dynamic Group Set

Ranganath Atreya, *Web Services Technologies, Amazon.com, Inc., Seattle, WA 98101 USA*

Neeraj Mittal, *Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA*

Sathya Peri, *Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA*

Abstract

The group mutual exclusion problem extends the traditional mutual exclusion problem by associating a type (or a group) with each critical section. In this problem, processes requesting critical sections of the same type can execute their critical sections concurrently. However, processes requesting critical sections of different types must execute their critical sections in a mutually exclusive manner.

We present a distributed algorithm for solving the group mutual exclusion problem based on the notion of *surrogate-quorum*. Intuitively, our algorithm uses the quorum that has been successfully locked by a request as a *surrogate* to service other *compatible* requests for the same type of critical section. Unlike the existing quorum-based algorithms for group mutual exclusion, our algorithm achieves a low message complexity of $O(q)$, where q is the maximum size of a quorum, while maintaining both synchronization delay and waiting time at two message hops. Moreover, similar to existing quorum-based algorithms, our algorithm has high maximum concurrency of n, where n is the number of processes in the system.

As opposed to some existing quorum-based algorithms, our algorithm can adapt without performance penalties to dynamic changes in the set of groups. Our simulation results indicate that our

A preliminary version of this paper has appeared earlier in 2005 Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS) [1].

This work was done while Ranganath Atreya was a student in the Department of Computer Science at The University of Texas at Dallas.

algorithm outperforms the existing quorum-based algorithms for group mutual exclusion by as much as 45% in some cases.

We also discuss how our algorithm can be extended to satisfy certain desirable properties such as concurrent entry and unnecessary blocking freedom. Further, we provide a generalized algorithm for group mutual exclusion based on the information structure of a mutual exclusion algorithm.

Index Terms

message-passing system, resource management, mutual exclusion, group mutual exclusion, quorum-based algorithm.

I. Introduction

Mutual exclusion is one of the most fundamental problems in concurrent systems including distributed systems. In this problem, access to a shared resource (that is, execution of critical section) by different processes must be synchronized to ensure its integrity by allowing at most one process to access the resource at a time. Numerous solutions [2], [3], [4], [5], [6] and extensions [7], [8], [9], [10] have been proposed to the basic mutual exclusion problem. More recently, another extension to the basic mutual exclusion problem, called *group mutual exclusion*, has been proposed [11]. In the group mutual exclusion problem, every critical section is associated with a type or a group. Critical sections belonging to the same group can be executed concurrently while critical sections belonging to different groups must be executed in a mutually exclusive manner.

The readers/writers problem can be modeled as a special case of group mutual exclusion using $n + 1$ groups, where n denotes the number of processes. In this case, all *read* requests belong to the same group and *write* requests by each process belongs to a different group. As another application of the problem, consider a CD-jukebox where data is stored on disks and only one disk can be loaded for access at a time [11]. In this example, when a disk is loaded, users that need data on the currently loaded disk can access the disk concurrently. While users that need data on different disks have to wait for the currently loaded disk to be unloaded.

Solutions for the group mutual exclusion problem have been proposed under both shared-memory and message-passing models. Solutions under shared-memory model can be found in [11], [12], [13], [14]. In this paper, we investigate the group mutual exclusion problem under message-passing model. For message-passing model, solutions to group mutual exclusion have been proposed for ring networks [15], [16] and tree networks [17]. Typically, solutions for ring and tree networks incur high synchronization delay and have high waiting time. For a fully connected network, two group mutual exclusion algorithms based on modification of the Ricart and Agrawala's algorithm for mutual exclusion [4] have been proposed in [18]. These algorithms have high message complexity of $O(n)$. The first algorithm has low expected concurrent of $O(1)$, whereas the second algorithm has high message overhead of $O(n)$.

The quorum-based mutual exclusion algorithm by Maekawa [19] has also been modified to derive two quorum-based algorithms for group mutual exclusion [20]. These algorithms use a special type of quorum system called the *group quorum system*. In a group quorum system, two quorums belonging to the same group need not intersect while quorums belonging to different groups must intersect. The maximum number of pair-wise disjoint quorums offered by a group quorum system is called the *degree* of the quorum system. In [20], Joung introduced a group quorum system called the *surficial quorum system*, which has a degree of $\sqrt{\frac{2n}{m(m-1)}}$, where m is the number of groups. When used with Maekawa's algorithm, the surficial quorum system can only allow up to the degree number of process of the same group to execute concurrently. To achieve unrestricted maximum concurrency, Joung also proposed two quorum-based algorithms, namely Maekawa_M and Maekawa_S, based on two separate modifications to the original Maekawa's quorum-based algorithm. The first modification enables quorum nodes to issue multiple locks for requests belonging to the same group. The draw-back of this approach is that, in the event of conflict between requests of different groups, deadlock resolution requires multiple locks to be taken back. This results in a high (worst-case) message complexity of $O(n \min\{m, \sqrt{n}\})$. To overcome this draw-back, Joung proposed a second modification that avoids deadlocks altogether. Specifically, deadlocks are avoided by locking quorum nodes in some fixed order. Although this approach reduces message complexity to $O(q)$, the synchronization delay evidently increases from two to $O(q)$ messages hops, where q is the maximum size of a quorum. In addition, both these algorithms need an *a priori* knowledge of the number of groups, which cannot change with time.

For some applications, the number of groups in the system may change dynamically during the course of execution. For instance, in the CD jukebox example, new CDs may be added at runtime. Hence it is desirable for a group mutual exclusion algorithm to be able to handle dynamic changes in the number of groups. Toyomura *et al* have proposed a quorum-based algorithm that uses a traditional quorum system [21]. Their algorithm is similar to Maekawa_M and therefore has high message complexity of $O(nq)$.

In this paper, we take a slightly different approach and introduce the notion of *surrogate-quorums*. The existing quorum-based algorithms [20] can either provide a low synchronization delay of two message hops or low message complexity of $O(q)$ but not both together. Hence, they are either inefficient or non-scalable. Our algorithm, on the other hand, achieves a low message complexity of $O(q)$ while maintaining both synchronization delay and waiting time at two message hops, thereby satisfying both of the seemingly opposing qualities—efficiency and scalability. To accomplish this, we introduce a relatively low message overhead of $O(b)$ per message, where b denotes the largest number of processes in whose quorum a node can belong to. Furthermore, unlike the existing quorum-based algorithms [20] which require a group quorum system, our algorithm assumes minimal properties of the underlying quorum system. This implies that our algorithm is decoupled from the underlying quorum system and more importantly does not need an *a priori* knowledge of the number of groups. In fact, our algorithm can adapt without performance penalties to dynamic changes (at runtime) in the number of groups. Finally, like the other quorum-based algorithms, the maximum concurrency is n. This implies that it is possible for all processes to execute their critical sections concurrently, provided all of them request critical sections of the same group.

We also explain how our algorithm can be extended to satisfy certain desirable properties such as *concurrent entry* [18] and *unnecessary blocking freedom* [22]. Further, we provide a generalized algorithm for group mutual exclusion based on the *information structure* of a mutual exclusion algorithm [23].

The rest of the paper is organized as follows. We present our system model and formally describe the group mutual exclusion problem in Section II. Section III provides the background information necessary to understand our algorithm. We then present our surrogate-quorum based algorithm for group mutual exclusion in Section IV. We prove its correctness in Section V, analyze its performance in Section VI and present our simulation results in Section VII. In Section VIII, we discuss several extensions to our basic algorithm. Finally, we present our conclusions in Section IX.

II. Model and Problem Definition

A. System Model

We assume an asynchronous distributed system comprising of a set of processes Π with $|\Pi| = n$. Processes communicate with each other by sending messages over a set of channels. We assume that there is a channel between every pair of processes. There is no global clock or shared memory. Processes are non-faulty. Channels are reliable and first-in-first-out (FIFO). Message delays are finite but may be unbounded. In this paper, we use lower case English letters (*e.g.*, x, y and z) to denote processes and upper case English letters (*e.g.*, P, Q and R) to denote sets of processes.

B. The Group Mutual Exclusion Problem

The problem of *group mutual exclusion (GME)* was first proposed in [11] as an extension to the traditional mutual exclusion problem. In this problem, every request for a critical section is associated with a type or a *group*. An algorithm for group mutual exclusion should satisfy the following properties:

- **group mutual exclusion:** at any time, no two processes, who have requested critical sections belonging to different groups, are in their critical sections simultaneously.
- **starvation freedom:** a process wishing to enter critical section succeeds eventually.

Clearly, any algorithm for solving the traditional mutual exclusion problem also solves the group mutual exclusion problem. However, such algorithms are sub-optimal because they force all critical sections to be executed in a mutually exclusive manner and therefore do not permit any concurrency whatsoever. To avoid such degenerate solutions and unnecessary synchronization, Joung proposed that an algorithm for achieving group mutual exclusion should satisfy the following desirable property:

- **concurrent entry (non-triviality):** if all requests are for critical sections belonging to the same group, then a requesting process should not be required to wait for entry into its critical section until some other process has left its critical section.

Intuitively, the concurrent entry property states that if processes currently in their critical sections, if any, never leave their critical sections and no process has or makes any conflicting request, then any process with a pending request should be eventually able to enter its critical section. The group mutual exclusion problem can be casted as *congenial talking philosophers* (CTP) problem [18]. In the CTP problem, a philosopher can be in one of the three states at any time: *thinking*, *waiting* and *talking*. Philosophers think alone but talk in fora. A philosopher, who is currently thinking, may at any time decide to join a specific *forum* by changing its state to waiting. Each forum is of a specific type and is held in a meeting room. There is only one meeting room available and at most one forum (of any type) can be in progress in the meeting room at any time. However, a forum can be attended by any number of philosophers. A philosopher, on entering the requested forum, changes its state and starts talking. The first philosopher to enter a forum *initiates* a *session* of that forum type and the last philosopher to leave it *terminates* that session.

Complexity Measures: To measure the performance of a group mutual exclusion algorithm, we use the following metrics:

- *message complexity:* the number of messages exchanged per request for critical section
- *synchronization delay:* the time elapsed between when the current forum terminates and when the next forum (of some other type) can commence
- *waiting time:* the time elapsed between when a process issues a request for critical section and when it actually enters the critical section
- *message overhead:* the amount of data piggybacked on a message (in terms of number of integers)
- *concurrency:* the number of processes that are in their critical sections at the same time

The first four metrics are used to evaluate the performance of a traditional mutual exclusion algorithm as well. The fifth metric is specific to a group mutual exclusion algorithm. Message-complexity and message-overhead determine the overhead imposed on the system by the group mutual exclusion algorithm at runtime. Synchronization delay is usually measured when the system is heavily loaded and there is a lot of contention among processes for access to the resource. Intuitively, synchronization delay and concurrency measure the system throughput that can be achieved when the system is heavily loaded. (The lower the synchronization delay and higher the concurrency, the greater is the system throughput.) One way to measure the concurrency of a group mutual exclusion is to determine the maximum number of processes that can execute their critical sections concurrently. This is referred to as the *maximum concurrency* of a group mutual exclusion algorithm. Waiting time captures the amount of time an application process has to wait for its request to be fulfilled. Waiting time is typically measured when the system is lightly loaded and, therefore, there is no contention for the resource.

III. Background

A. A Quorum System

A *quorum* is a subset of nodes or processes. Although nodes and processes are identical, following the convention in [20], we use the term node specifically when referring to the role of a process as a quorum member. A *quorum system* C*,* also referred to as a *coterie,* for (traditional) mutual exclusion is a set of quorums satisfying the following properties:

- **intersection:** $\forall P, Q \in C :: P \cap Q \neq \emptyset$
- **minimality:** $\forall P, Q \in C : P \neq Q : P \not\subseteq Q$

If a process enters its critical section only after it has successfully locked all nodes in some quorum, then the intersection property ensures that no two processes can execute their critical sections concurrently. The minimality property ensures that no process is required to lock more nodes than necessary to achieve mutual exclusion. For a node x, let B_x denote the set of processes that can send requests to x. If a process randomly chooses its quorum, then the size of B_x can be as large as n. However in our algorithm, we assume that each process is assigned a fixed quorum and so $B_x = \{Q \in C | x \in Q\}$. For a grid quorum system, $|B_x| = O(\sqrt{n})$. Henceforth, we refer to B_x as the *membership set of* x. In addition to the intersection and minimality properties, it is desirable that the coterie C also satisfy the following properties:

- $\forall P, Q \in C :: |P| = |Q|$
- $\forall x, y \in \Pi :: |B_x| = |B_y|$

Existing quorum-based algorithms [20] for group mutual exclusion use a special type of quorum system called a group quorum system. In a group quorum system, any two quorums belonging to the same group need not intersect while quorums belonging to different groups must intersect. We do not use a group quorum system in this paper; instead we employ a traditional quorum system. Henceforth, the term “quorum system” is used to refer to the traditional quorum system.

B. The Maekawa’s Algorithm

Maekawa’s algorithm implements mutual exclusion by using a coterie that satisfies the aforementioned properties. Lamport’s logical clock [3] is used to assign a timestamp to every request for critical section. A request with a smaller timestamp has a *higher priority* than a request with a larger timestamp (ties are broken using process identifiers). Maekawa’s algorithm works as follows:

1) When a process wishes to enter critical section, it selects a quorum and sends a REQUEST message to all the quorum members. It enters the critical section once it has successfully locked all its quorum members. On leaving the critical section, the process unlocks all its quorum members by sending a RELEASED message.
2) A node, on receiving a REQUEST message, checks to see whether it has already been locked by some other process. If not, it grants the lock to the requesting process by sending a LOCKED message to it. Otherwise, the node uses timestamps to determine whether the process currently holding a lock on it—hereafter referred to as the *locking process*—should be preempted. In case the node decides not to preempt the locking process, it sends a

FAILED message to the requesting process. Otherwise, it sends an INQUIRE message to the locking process.

3) A process, on receiving an INQUIRE message from a quorum node, unlocks the quorum node by sending a RELINQUISH message as and when it realizes that it will not be able to successfully lock all its quorum members. This is ascertained when a FAILED message is received from one of the quorum members.
4) A node, on receiving a RELINQUISH or RELEASED message, grants the lock to the process whose request has the highest priority among all the pending requests, if any.

Maekawa [19] prove that the message complexity of the above algorithm is $O(q)$, where q is the maximum size of a quorum. Further, its (best-case) synchronization delay and waiting time are both two message hops. (When analyzing the synchronization delay of a quorum-based algorithm derived from the Maekawa's algorithm, we ignore the delay incurred due to deadlock resolution and only analyze the *best-case* synchronization delay. This is consistent with the practice used by other researchers [19], [20].)

IV. A Surrogate-Quorum based Algorithm

We now describe our approach for solving the group mutual exclusion problem. We call two requests as *compatible* if they are for the same type; otherwise they are said to be *conflicting*.

A. The Main Idea

The main focus of our algorithm is to provide the following advantages. Our algorithm should be scalable and hence achieve low message complexity and low message overhead. To that end, we choose a quorum-based approach. Our algorithm should be efficient, which means that it should have low waiting time, low synchronization delay and high maximum concurrency. In addition, we want our algorithm to be independent of the underlying quorum system and be able to handle dynamic changes, at runtime, in the number of groups. Therefore, unlike the existing quorum-based algorithms [20], we do not assume a group quorum system. On the contrary, we assume minimal properties for the underlying quorum system. Particularly, we only assume the properties listed in Section III-A.

One approach to achieving concurrency is by enabling nodes to issue multiple locks [20]. However, in this approach deadlock resolution may require multiple locks to be preempted thereby increasing message complexity. To ensure scalability, we take the *leader-follower* approach introduced in [18] along with the notion of surrogate-quorum. In our approach, processes requesting entry into their critical sections try to lock their respective quorums. A process that successfully captures its quorum *invites* other processes with compatible request to enter the forum. Therefore a process can enter the forum by either locking all its quorum members or by receiving an invitation from another process. The process in the former case is called a *leader* and that in the latter case is called a *follower*. In order to inform a leader about other requests, a quorum member on sending its lock also sends compatible requests that are currently in its queue. To ensure group mutual exclusion property, the leader does not release its quorum until all its followers have left the forum. We therefore use the quorum of the leader as a surrogate for its followers and hence the name *surrogate-quorum*. To avoid repetition, we only describe our extensions to Maekawa's algorithm.

1) A node, when sending a LOCKED message to a process, piggybacks all requests currently in its queue that are compatible with the request by the locking process.
2) A process, on receiving a LOCKED message, stores all the requests that were piggybacked on the message. Once it has successfully locked all its quorum members, it sends an INVITE message to processes who made these requests.
3) A process, on receiving an INVITE message for its current request, unlocks all its quorum members by sending a CANCEL message. It then enters the forum.
4) A node, on receiving a CANCEL message from a process, removes its request from the queue, if it exists.
5) Once a follower exits the forum, it sends a LEAVE message to its leader.
6) A leader maintains the lock on its quorum members until it has received a LEAVE message from all its followers and has itself left the forum. It then sends a RELEASED message to its quorum members.
7) A node, on receiving a RELEASED message from a process, removes all those requests from its queue that it piggybacked on the last LOCKED message it sent.

Since processes can enter a forum (as a follower) without locking all its quorum members, fulfilled requests may persist in the system for some time. We refer to these requests and the messages generated due to these requests as "stale". A process may receive stale LOCKED, FAILED and INVITE messages due to its stale requests. Each of these messages can be piggybacked with the timestamp of the request that generated them. As a result, the requesting process upon receiving a message can easily determine whether the message is stale. A process needs to send a LEAVE message to the leader that sent a stale INVITE message to it. Stale LOCKED and FAILED messages should be ignored.

If a leader exits its forum but has not received a LEAVE message from all its followers, then the leader is called a *surrogate-leader*. It should be noted that, a process in the surrogate-leader mode *can execute its underlying program unimpeded*. With the above modification, our algorithm has a message complexity of $O(q)$ and a synchronization delay of three message hops (LEAVE, RELEASED and LOCKED). Since a node can piggyback at most one compatible request per process in its membership set, the message overhead of LOCKED messages is $O(b)$, where b is the size of the largest membership set.

B. Reducing Synchronization Delay

Synchronization delay has a significant impact on efficiency, especially system throughput. Therefore, it is desirable to reduce it further. It is evident that LEAVE messages can be eliminated by allowing a follower to directly release the quorum members of its leader. To do so and still ensure group mutual exclusion property, we make the following modifications:

1) A leader upon entering a forum sets its weight to one.
2) Upon sending an INVITE message a leader reduces its current weight by half and piggybacks the other half over the INVITE message.
3) A leader upon exiting its forum, instead of waiting for LEAVE messages, sends a RELEASED message along with its remaining weight to its quorum members.
4) A follower upon exiting its forum, instead of sending a LEAVE message to the leader, sends a RELEASED message along with the weight it received over the INVITE message to all quorum members of its *leader*.
5) A node accumulates all the weights it received over RELEASED messages and maintains its lock until its cumulative weight becomes one.

An efficient solution for weight distribution and recovery was proposed in [24]. The proposed approach involved storing rational numbers as integers in fractional form thus avoiding real numbers. Using this approach, we only incur an overhead of two integers per INVITE and RELEASED message.

It is clear that a fulfilled request may receive at most q stale INVITE messages, one from each quorum member. Before this modification, for each stale INVITE message, a node only sent one LEAVE message and hence the message complexity was $O(q)$. However, according to the above modifications, for each stale INVITE message a node sends q RELEASED messages thereby increasing the message complexity to $O(q^2)$. To ensure scalability, we propose another modification that reduces message complexity to $O(q)$ while maintaining the message overhead at $O(b)$.

C. Avoiding Stale INVITE Messages

To lower the message complexity to $O(q)$, we need to eliminate stale INVITE messages. It is clear that a quorum member sends LOCKED message only after receiving RELEASED messages from all processes in the current forum. The "new" leader, due to the intersection property, has to obtain LOCKED message from at least one quorum member of an "old" leader. Hence there exists a causal path from all processes leaving a forum to a process entering a forum later as a leader. In this modification, we basically exploit this causal path to pass on information about stale requests to the next leader. We propose the following changes:

1) Each process x maintains a list of timestamps; there is one entry in the list for each process in the system. The entry for process y in the list contains the timestamp of the latest request by y that has been fulfilled according to x's knowledge.
2) A node, on receiving a RELEASED message from a process updates its list with the timestamp of the request that sent the RELEASED message. If the list already contains a timestamp from the same process then the latest timestamp is retained.
3) Upon sending a LOCKED message to a process, a node in addition to piggybacking compatible requests, also piggybacks those timestamps from its list that have not been previously sent to the process.
4) A process, on receiving a LOCKED message from a node, updates its list using the timestamps (for stale requests) received along with the LOCKED message. (Note that the list needs to be updated even if the LOCKED message is stale.) Again, if the list already contains a timestamp from the same process, then the latest timestamp is retained.
5) Upon successfully locking all quorum members, a leader desists from sending INVITE message to a process y whose request has a timestamp that is less than or equal to the corresponding entry in its vector.

With the above modifications, we have an algorithm that has a synchronization delay of two message hops (RELEASED followed by LOCKED) and a message complexity of $O(q)$. However, we appear to have increased the message overhead of LOCKED messages. Since every process in the system may concurrently enter a given forum, in the worst-case the message overhead of a LOCKED message may be $O(n)$. We show later that this overhead in fact amortizes to $O(b)$ over all messages.

D. Formal Description of Our Algorithm

We refer to our algorithm as Surrogate. A formal description of Surrogate can be found in Figure 1 and Figure 2. Due to lack of space, some actions of Surrogate which are identical to those of the Maekawa's algorithm (such as those used for deadlock avoidance), have been omitted. Also, every message is tagged with the timestamp of the request for which it is being exchanged along with other information (*e.g.*, forum type requested). This is not shown explicitly in the formal description due to lack of space.

```
Actions for process $x$:

Variables:
    $staleList$:  list of latest stale requests for each process as per $x$'s knowledge;
    $leader$:  the leader of $x$'s forum (defined only if currently in forum);
    $weight$:  weight to be sent back to quorum members (defined only if currently in forum);
    $inviteSet$:  processes to which an INVITE message to be sent (defined only if there is
                  a currently pending request);

● On generating a request for forum:
    send REQUEST message to all quorum members;
    $inviteSet := \emptyset$;

● On receiving a LOCKED($compatibleSet, newEntries$) message from node $y$:
    update $staleList$ using $newEntries$;
    update $inviteSet$ using $compatibleSet$ and $staleList$;
    if (all quorums members have been locked) then
        $leader,\ weight := x,\ 1$;
        for each request in $inviteSet$ do
            $weight := weight/2$;
            send INVITE($weight$) message to the requesting process;
        endfor;
        enter the forum;
    endif;

● On receiving an INVITE($weightRcvd$) message from process $y$:
    $leader,\ weight := y,\ weightRcvd$;
    send CANCEL message to all quorum members;
    enter the forum;

● On leaving a forum:
    send RELEASED($weight$) message to all members of $leader$'s quorum;
```

Fig. 1. Formal description of Surrogate for a process.

```
Actions for node $x$:

Variables:
        $staleList$:  list of latest stale requests for each process as per $x$'s knowledge;
        $weight$:  weight collected so far (defined only if currently locked by some process);

● On deciding to send a LOCKED message to process $y$:
        $weight := 0$;
        $compatibleSet$ := set of requests in the queue that are compatible with $y$'s request;
        $newEntries$ := set of entries in $staleList$ that have changed since the last LOCKED
                        message sent to $y$;
        send LOCKED($compatibleSet, newEntries$) message to $y$;

● On receiving a CANCEL message from process $y$:
        if ($y$ is not the current lock holder) then
                update $staleList$ and remove $y$'s request from the queue;
        else
                treat it like a RELEASED message carrying a weight of $1$;
        endif;

● On receiving a RELEASED($weightRcvd$) message from process $y$:
        update $staleList$ and remove $y$'s request from the queue, if present;
        $weight := weight + weightRcvd$;
        if ($weight = 1$) then
                // the lock has been released
                <do what Maekawa's algorithm would do here>;
        endif;
```

Fig. 2. Formal description of Surrogate for a node.

V. Proof of Correctness

In this section, we formally prove that Surrogate in fact satisfies the first two properties of a group mutual exclusion algorithm, namely group mutual exclusion and starvation freedom. In Section VIII, we describe a simple extension to Surrogate that achieves concurrent entry.

Note that a request for critical section may have to wait for one or more requests to release *locks* on its quorum members. These wait relationships may recursively grow to form what we call *wait-for-subgraphs* emanating from a request. Wait-for-subgraphs may branch out and form more wait-for-subgraphs. However, as we show later, these wait-for-subgraphs eventually unlink.

In the following proofs, we model the execution of the system as an infinite alternating sequence of global states and events, $\sigma = S_0 e_1 S_1 \ldots S_{i-1} e_i S_i \ldots$. For executions with finite number of such global states, we assume the existence of a hypothetical *nop* event for no-

$\mathcal{R}_i$	$\triangleq$	set of all requests for critical section made in some global state up to (and including) global state S_i
$\alpha.proc$	$\triangleq$	process to which request α belongs
$type(\alpha)$	$\triangleq$	type of request α
$quorum(\alpha)$	$\triangleq$	quorum selected by request α
$forum(\alpha, i)$	$\triangleq$	$\alpha.proc$ is in a forum at global state S_i and α is the latest request of $\alpha.proc$
$leader(\alpha)$	$\triangleq$	request α is satisfied as a leader (assuming α is eventually satisfied)
$follower(\alpha)$	$\triangleq$	request α is satisfied as a follower (assuming α is eventually satisfied)
$weight(x, i)$	$\triangleq$	weight of node x at global state S_i
$locked(x, \alpha, i)$	$\triangleq$	node x has sent its lock to request α but has not recovered some part of it at global state S_i, that is, $\langle \exists j : j \leq i : (x \text{ sent a LOCKED message to } \alpha \text{ at global state } S_j) \wedge \langle \forall k : j < k \leq i : (x \text{ did not send a LOCKED message at global state } S_k) \wedge (weight(x, k) < 1)\rangle\rangle$

Fig. 3. Notation used in the safety proof.

operation that remains enabled in all states following the last global state. A *nop* event does nothing and so in our model for executions with finite sequence of global states, the final state is repeated infinitely. The system is assumed to transition from a global state S_{i-1} on executing an *enabled* event e_i to state S_i. We assume that a continuously enabled is eventually executed.

In this paper, we use lower case Greek letters (*e.g.*, α, β and γ) to denote requests for critical section. When a node x sends a message MSG (LOCKED, INQUIRE or FAILED) to a process y for a request α, we say that x sent MSG to α (or, equivalently, α received MSG from x). Likewise, when a process x sends a message MSG (RELINQUISH or RELEASED) to a node y for request α, we say that α sent MSG to y.

A. Proving Safety (Group Mutual Exclusion)

For proving the safety property, we use the notation described in Figure 3. Using the notation in the figure, we state some properties about our algorithm, which can be verified easily. The first property states that if a request is currently being satisfied, then there exists a request of the same type that currently holds locks on all its quorum members.

$$forum(\alpha, i) \ \Rightarrow\ \langle \exists \beta : \beta \in \mathcal{R}_i : leader(\beta) \ \wedge\ (type(\alpha) = type(\beta)) \wedge \langle \forall\, x : x \in quorum(\beta) : locked(x, \beta, i)\rangle\rangle \tag{1}$$

The second property states that a node cannot be locked by two different requests at the same time. Formally,

$$locked(x, \alpha, i) \wedge locked(x, \beta, i) \;\Rightarrow\; \alpha = \beta \tag{2}$$

We now prove that our algorithm is safe.

Theorem 1 (safety property): Surrogate satisfies the group mutual exclusion property, that is, two requests of different types cannot be satisfied concurrently. Formally,

$$forum(\alpha, i) \wedge forum(\beta, i) \;\Rightarrow\; type(\alpha) = type(\beta)$$

Proof: Assume that $forum(\alpha, i)$ and $forum(\beta, i)$ hold. Since $forum(\alpha, i)$ holds, using (1), there exists a request $\gamma \in \mathcal{R}_i$ such that:

$$leader(\gamma) \wedge (type(\alpha) = type(\gamma)) \wedge \langle \forall\, x : x \in quorum(\gamma) : locked(x, \gamma, i) \rangle \tag{3}$$

Likewise, since $forum(\beta, i)$ holds, using (2), there exists a request $\delta \in \mathcal{R}_i$ such that:

$$leader(\delta) \wedge (type(\beta) = type(\delta)) \wedge \langle \forall\, y : y \in quorum(\delta) : locked(y, \delta, i) \rangle \tag{4}$$

From the intersection property of a quorum system, there is node z in $quorum(\gamma) \cap quorum(\delta)$ such that z has been locked by both γ and δ in global state S_i. Observe that a node can be locked by at most one request in any global state. This means that γ and δ are actually same requests. From (3) and (4), $type(\alpha) = type(\gamma) = type(\delta) = type(\beta)$. ■

B. Proving Liveness (Starvation Freedom)

Our algorithm for group mutual exclusion is derived from Maekawa's algorithm for mutual exclusion. Maekawa himself gave a very informal proof for liveness in his paper [19]. We, on the other hand, provide a more formal proof for starvation freedom of our algorithm in this paper.

We say that a request is satisfied as soon as the requesting process enters the critical section because of the request. For proving liveness, we use the notation described in Figure 4 in addition to the notation used in the safety proof.

Before giving the proof, we formalize the concept of "waiting-on relation" and "wait-for-graph". We say that a request α is *waiting on* another request β if (1) α is still pending, (2) the timestamp of α is smaller than the timestamp of β (ties are broken using process identifiers), and (3) α is waiting for a lock that was sent to β. Formally,

$$\begin{aligned} waiting\text{-}on(\alpha, \beta, i) \;\triangleq\;& pending(\alpha, i) \;\wedge\; (ts(\alpha) < ts(\beta)) \;\wedge \\ & \langle \exists x : x \in quorum(\alpha) \cap quorum(\beta) : (\alpha \in queue(x, i)) \wedge locked(x, \beta, i) \rangle \end{aligned}$$

$ts(\alpha)$	$\triangleq$	timestamp of request α
$queue(x, i)$	$\triangleq$	set of requests that belong to the request queue of node x at global state S_i
$failcount(\alpha, i)$	$\triangleq$	number of FAILED messages received by α until (and including) global state S_i
$pending(\alpha, i)$	$\triangleq$	request α has not been satisfi ed until (and including) global state S_i, that is, $\langle \forall\, j : j \leq i : \alpha \in \mathcal{R}_j \;\Rightarrow\; \neg forum(\alpha, j) \rangle$

Fig. 4. Additional notation used in the liveness proof.

Based on the above definition of waiting-on relation, each state of a distributed system running Surrogate can be represented by a directed graph. For any system state, the set of vertices in the graph is given by the set of requests that have been made so far and a directed edge exists from vertex α to vertex β if request α is waiting on request β in the given state. The graph for a state S_i is called the *wait-for-graph* at state S_i and is denoted by $WFG(i)$.

Lemma 2: $WFG(i)$ satisfies the following properties:

1) $WFG(i)$ is acyclic.
2) All paths in $WFG(i)$ are simple paths.
3) If α has been satisfied in S_i, then α does not have any outgoing edges in $WFG(i)$.
4) The maximum length of all paths in $WFG(i)$ is bounded by $|\Pi|$.

Proof: The first property follows from the fact that the timestamp value along any path increases monotonically. The second property follows from the first property. The third property follows from the definition of waiting-on relation (for a request to have an outgoing edge, it should still be pending). Finally, the fourth property follows from the third property and the fact that a process does not generate the next request until its current request has been fulfilled. ■

For a global state S_i, the subgraph of $WFG(i)$ emanating from a request α is called the *wait-for-subgraph* of α at S_i. We define the *wait-set* of a α at S_i, denoted by $wait\text{-}set(\alpha, i)$, as the set of requests that α is directly waiting on at S_i, that is, $wait\text{-}set(\alpha, i) = \{ \beta \mid waiting\text{-}on(\alpha, \beta, i) \}$. The wait-set of a fulfilled request is trivially empty. Note that, once every node in $quorum(\alpha)$ has received the request α, the wait-set of α cannot grow. Formally,

$$pending(\alpha, i) \wedge \langle \forall\, x : x \in quorum(\alpha) : \alpha \in queue(x, i) \rangle \;\Rightarrow \\ \langle \forall\, r, s : r, s \geq i : r \leq s \;\Rightarrow\; wait\text{-}set(\alpha, r) \supseteq wait\text{-}set(\alpha, s) \rangle \tag{5}$$

Also, note that, for a request to be satisfied as a leader, its wait-set should eventually become permanently empty. This is relatively easy to prove if a request sends a RELINQUISH message immediately on receiving an INQUIRE message (that is, without first waiting to receive a FAILED message). However, the proof is more involved when FAILED messages are also considered.

In our proof, we use two attributes of a pending request, namely *potence* and *omnipotence*, which are defined as follows. A pending request α is said to be *potent* in a global state S_i, denoted by $potent(\alpha, i)$, if no request with higher priority (that is, smaller timestamp) is ever generated in a global state following S_i. Further, a potent request α is said to be *omnipotent* in S_i, denoted by $omnipotent(\alpha, i)$, if it has the highest priority among all pending requests in S_i.

Our liveness proof consists of the following steps. First, we show that the wait-set of a pending request eventually becomes permanently empty. Second, we show that once a request becomes omnipotent and its wait-set becomes permanently empty it is eventually satisfied. Finally, we show that a pending request eventually becomes potent and then omnipotent.

1) Wait-set eventually becomes permanently empty: A process that has entered a forum as a leader may send INVITE messages to other processes. If a process receives an INVITE message for a request that has already been fulfilled, it simply ignores the message. If that happens, quorum members of the leader will not be able recover their locks, and no request will be fulfilled thereafter. Therefore we first show that INVITE messages are never sent to stale requests.

Lemma 3: Surrogate does not generate any stale INVITE messages.

Proof: From the intersection property of a quorum system, there exists a causal path via RELEASED and LOCKED messages from all requests satisfied in some forum to the leader of every subsequent forum. Through this causal path, the stale requests vector of a leader is updated with the requests that have already been satisfied in all earlier forums. Therefore a leader does not send any stale INVITE message. ■

We now show that quorum members of a satisfied request eventually recover their locks.

Lemma 4: Once a request has been satisfied, all its quorum members eventually recover their locks. Formally,

$$forum(\alpha, i) \land locked(x, \alpha, i) \;\Rightarrow\; \langle \exists\, j : j \geq i : \neg locked(x, \alpha, j) \rangle$$

Proof: There are two cases to consider depending on whether request α is satisfied as a leader or a follower.

First, assume the former. Note that a leader has only a finite number of followers. The leader, on leaving the forum, sends its weight to all its quorum members via a RELEASED message. Further, each of its followers, on leaving the forum, sends the weight it received through an INVITE message to the members of its leader's quorum, again via a RELEASED message. A quorum member recovers its lock once it has received all these RELEASED messages,

Now, assume the latter. A follower, on receiving an INVITE message, sends a CANCEL message to all its quorum members. A quorum member recovers its lock once it has received the CANCEL message. ■

The next two lemmas establish that no edge in a wait-for-graph is *permanent*.

Lemma 5: If a request eventually either receives a FAILED message or is satisfied, then no request can (directly) wait on it forever. Formally,

$$(\beta \in \textit{wait-set}(\alpha, i)) \ \wedge \ \Big(\textit{forum}(\beta, i) \ \vee \ (\textit{failcount}(\beta, i) > 0)\Big) \ \Rightarrow \ \langle \exists\, j : j \geq i : \beta \notin \textit{wait-set}(\alpha, j) \rangle$$

Proof: Consider a node $x \in \textit{quorum}(\alpha) \cap \textit{quorum}(\beta)$ such that when x receives the request α it had granted its lock to the request β. Clearly, x, on receiving α, sends an INQUIRE message to *beta*, if it has not done so already. On receiving the INQUIRE message, β waits until it either receives a FAILED message or is satisfied. By assumption, one of the two conditions eventually holds. In case the first condition holds before the other, β sends a RELINQUISH message to node x. Otherwise, in case the second condition holds, from Lemma 4, x eventually recovers its lock that it had granted to β. Therefore, by universal generalization, eventually every quorum member of α that had granted the lock to β recovers its lock. Once that happens, β leaves the wait-set of α. ■

We define the *level* of a request α in a global state S_i, denoted by $\textit{level}(\alpha, i)$, as the maximum length of any path starting from α in $WFG(i)$. From Lemma 2, the level of a request is upper bounded by $|\Pi|$. Further, we define the *rank* of α in S_i, denoted by $\textit{rank}(\alpha, i)$, as the maximum value attained by its level in any global state including and following S_i. Formally,

$$\textit{rank}(\alpha, i) \ \triangleq \ \max_{j \geq i} \{ \ \textit{level}(\alpha, j) \ \}$$

Note that, the rank of a request is monotonically non-increasing unlike its level, which is not. We use the notion of rank to prove the following lemma.

Lemma 6: Every request eventually either receives a FAILED message or is satisfied. Formally,

$$\textit{pending}(\alpha, i) \Rightarrow \langle \exists j : j \geq i : \ (\textit{failcount}(\alpha, j) > 0) \ \vee \ \textit{forum}(\alpha, j) \rangle$$

Proof: Consider a request that never receives a FAILED message. Clearly, when it is inserted in the queue of its quorum member, it is inserted at the front. Further, it continuously stays at the front at least until the quorum member sends a LOCKED message to it. As a result, once its wait-set becomes permanently empty, it eventually receives LOCKED messages from all its quorum members and gets satisfied. Therefore, we can conclude that once the wait-set of a request becomes permanently empty, it either receives a FAILED message or is satisfied. Formally,

$$\langle \forall\, r : r \geq i : \textit{wait-set}(\alpha, r) = \emptyset \rangle \ \Rightarrow \ \langle \exists\, j : j \geq i : (\textit{failcount}(\alpha, j) > 0) \vee \ \textit{forum}(\alpha, j) \rangle \qquad (6)$$

Now, the proof of the lemma is by induction on the rank of a request.

- **Base Case ($\mathbf{rank}(\alpha, \mathbf{i}) = \mathbf{0}$):** Clearly, $\textit{wait-set}(\alpha, r)$ is empty for all $r \geq i$. Using (6), the property holds.
- **Induction Step ($\mathbf{rank}(\alpha, \mathbf{i}) = \mathbf{k}$ with $\mathbf{k} > \mathbf{0}$):** From (5), eventually the system reaches a global state S_r after which the wait-set of α stops growing. Consider a request β in $\textit{wait-set}(\alpha, r)$. We claim that eventually β leaves the wait-set of α. Assume the contrary, that is, β never leaves the wait-set of α. This implies that $\textit{rank}(\beta, r) < \textit{rank}(\alpha, r) \leq \textit{rank}(\alpha, i)$. By induction hypothesis, β eventually either receives a FAILED message or is satisfied. From Lemma 5, β eventually leaves the wait-set of α. By repeatedly using this argument, we can conclude the wait-set of α eventually becomes empty. Therefore, using (6), the property holds.

This establishes the lemma. ■

Combining (5), Lemma 5 and Lemma 6, we can conclude that eventually the wait-set of a request becomes permanently empty. Formally,

$$\textit{pending}(\alpha, i) \;\Rightarrow\; \langle \exists\, j : j \geq i : \langle \forall\, r : r \geq j : \textit{wait-set}(\alpha, r) = \emptyset \rangle\rangle \tag{7}$$

2) An omnipotent request is eventually satisfied:

Lemma 7: Every omnipotent request is eventually satisfied. Formally,

$$\textit{omnipotent}(\alpha, i) \Rightarrow \langle \exists\, j : j \geq i : \textit{forum}(\alpha, j)\rangle$$

Proof: From (7), system eventually reaches a state in which the wait-set of α is empty and stays empty thereafter. Consider a quorum member x of α. Any request that is before α in the queue of x is a stale request. Clearly, all stale requests are eventually removed from the queue. As a result, α eventually reaches the front of x's queue and, moreover, stays in the front. Once that happens, x sends a LOCKED message to α and never sends an INQUIRE message after that. By universal generalization, every quorum member of α eventually sends a LOCKED message to α and never sends an INQUIRE message after that. On receiving these LOCKED messages from all its quorum members, α enters the forum and is satisfied. ■

3) A pending request eventually becomes omnipotent: We first show that a pending request eventually becomes potent (unless it is satisfied).

Lemma 8: Every pending request eventually becomes potent or is satisfied. Formally,

$$\textit{pending}(\alpha, i) \;\Rightarrow\; \langle \exists\, j : j \geq i : \textit{potent}(\alpha, j) \vee \textit{forum}(\alpha, j)\rangle$$

Proof: Assume that α is never satisfied. Thus we need to show that it eventually becomes potent. We say that a pending request α is potent *with respect to* a process x in a global state

S_i if x never generates a request with higher priority than α in any global state following S_i. Clearly, to prove the lemma, it suffices to show that α eventually becomes potent with respect to every process in the system.

Consider a process x. There are two cases to consider: after S_i, x either generates an infinite number of requests or generates only a finite number of requests. First, assume the former. Note that the logical clock value at x strictly increases every time x generates a request. As a result, eventually every request generated by x has a higher timestamp than α and therefore α eventually becomes potent with respect to x. Now, assume the latter. In this case, α becomes potent with respect to x as soon as x generates its last request. ■

We now show that a potent request eventually becomes omnipotent (unless it is satisfied).

Lemma 9: Every potent request eventually becomes omnipotent or is satisfied. Formally,

$$potent(\alpha, i) \;\Rightarrow\; \langle \exists j : j \geq i : omnipotent(\alpha, j) \vee forum(\alpha, j) \rangle$$

Proof: We define the *compete-set* of a pending request α in a global state S_i, denoted by *compete-set*(α, i), as the set of all pending requests in global state S_i that have higher priority than α. Formally,

$$compete\text{-}set(\alpha, i) \triangleq \{ \beta \mid pending(\beta, i) \;\text{ and }\; ts(\beta) < ts(\alpha) \}$$

Assume that α is potent in S_i and is never satisfied. Note that the compete-set of α cannot grow after S_i. Formally,

$$\langle \forall r, s : r, s \geq i : r \leq s \;\Rightarrow\; compete\text{-}set(\alpha, r) \supseteq compete\text{-}set(\alpha, s) \rangle \tag{8}$$

Therefore it suffices to prove that, starting from any global state following (and including) S_i, the compete set of α eventually shrinks. Formally,

$$\langle \forall r : r \geq i : \langle \exists s : s > r : compete\text{-}set(\alpha, r) \supsetneq compete\text{-}set(\alpha, s) \rangle \rangle \tag{9}$$

By applying (9) repeatedly, we can show that the compete-set of α eventually becomes empty. At that time, α becomes omnipotent. To prove (9), observe that, once a request becomes potent, it stays potent until it is satisfied. Moreover, if α is potent in S_r, then every request in *compete-set*(α, r) is also potent in S_r. Consider the request β with the smallest timestamp in *compete-set*(α, r). Clearly, β is omnipotent in S_r. Using Lemma 8, β is eventually satisfied and therefore eventually leaves the compete set of α. ■

4) Combining the three: It follows from Lemmas 7–9 that:

Theorem 10 (liveness property): Every request is eventually fulfilled. Formally,

$$pending(\alpha, i) \;\Rightarrow\; \langle \exists j : j \geq i : forum(\alpha, j) \rangle$$

VI. Complexity Analysis

In this section we analyze the performance of our algorithm with respect to the following metrics: message complexity, message overhead amortized over all messages, synchronization delay and maximum concurrency. As usual, q denotes the maximum size of a quorum.

Theorem 11: The worst-case message complexity of Surrogate is $O(q)$.

Proof: For each type of message, we count the maximum number of messages that are exchanged of that particular type due to a given request. Evidently, the number of REQUEST, FAILED and CANCEL messages are bounded by q each. We call a LOCKED message from a quorum node as *successful* if the locking request never sends a RELINQUISH message to the quorum node after receiving that LOCKED message. Clearly, the number of successful LOCKED messages is bounded by q. An INQUIRE message is generated only when a *new* request arrives at a quorum node and it is never generated for an old request. We charge an INQUIRE message to the new request on whose behalf the INQUIRE message was generated. Therefore the number of INQUIRE, RELINQUISH and unsuccessful LOCKED messages are each bounded by q per request. All that remains to be bound are the number of INVITE and RELEASED messages. Since a follower upon exiting its forum, sends RELEASED messages to all nodes in its leader's quorum, the number of RELEASED messages is equal to q times the number of INVITE messages. From Lemma 4, a process can receive at most one INVITE message per request because no INVITE messages are sent for stale requests. Hence the number of RELEASED messages is bounded by q per request. ■

We now bound the worst-case message overhead of Surrogate. It is clear that all messages except LOCKED messages have an overhead of $O(1)$. The worst-case overhead of LOCKED messages is $O(n)$. For most systems, it is better to exchange fewer number of large messages than a large number of smaller messages. Notwithstanding this fact, our algorithm has a low amortized message overhead of $O(b)$ over all messages, where b denotes the size of the largest membership set.

Theorem 12: The message overhead of Surrogate is $O(b)$ amortized over all messages.

Proof: All messages except LOCKED messages have an overhead of $O(1)$. Now, LOCKED messages carry two separate kinds of overhead, namely overhead due to compatible requests and overhead due to stale requests. Since a node can only receive requests from its membership set, the number of compatible requests piggybacked over a single LOCKED message is bounded by b per request. We now bound the overhead due to stale requests. Every node only piggybacks the timestamp of a stale request once to each process in its membership set. Every time the timestamp of a stale request is piggybacked, we charge it towards the stale request. Since a node can only send LOCKED messages to at most b nodes, each request can be piggybacked on at most $O(qb)$ LOCKED messages as a stale request. Amortizing the overhead over at least q messages that are exchanged on behalf of a request, we obtain an amortized message overhead of $O(b)$. ■

It is evident that, in a lightly loaded system, on generating a request, a process can enter its forum in two message hop (REQUEST and LOCKED) delays. In a heavily loaded system, a process can enter its forum in two message hop (RELEASED and LOCKED) delays after another process leaves its forum.

Theorem 13: The synchronization delay and the waiting time for Surrogate are both two message hops.

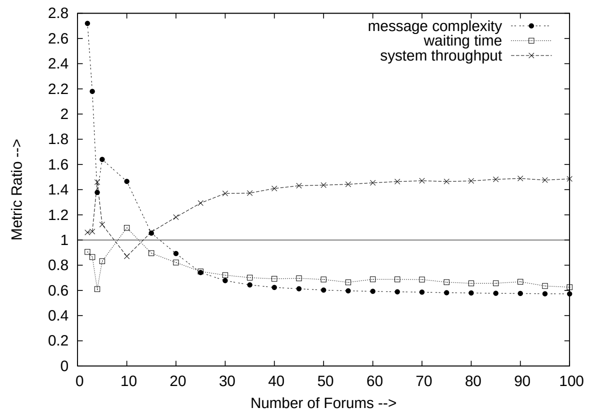
Clearly, if all processes make compatible requests, then all of them can be in the forum at the same time.

Theorem 14: The maximum concurrency of Surrogate is n.

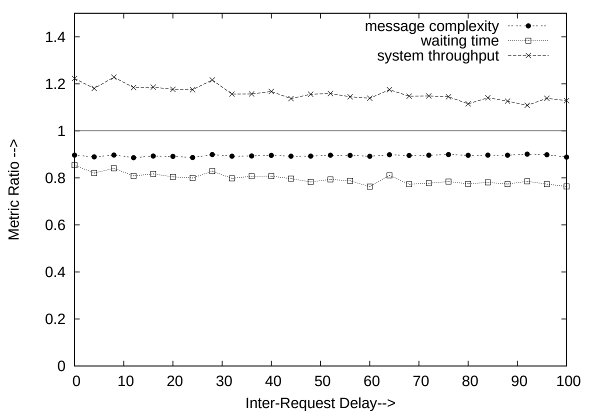
VII. Experimental Evaluation

We experimentally compare the performance of Surrogate with Joung's first algorithm Maekawa_M using a discrete-event simulation. We compare the performance of the two algorithms with respect to three metrics, namely message complexity, waiting time and system throughput. To make it easier to compare the two algorithms, we report the ratio $\frac{\text{Surrogate's performance}}{\text{Maekawa\_M's performance}}$ for each metric. Note that, for message complexity and waiting time, a ratio of less than one would imply that Surrogate has better performance than Maekawa_M. On the other hand, for system throughput, a ratio of greater than one would imply that Surrogate has better performance than Maekawa_M.

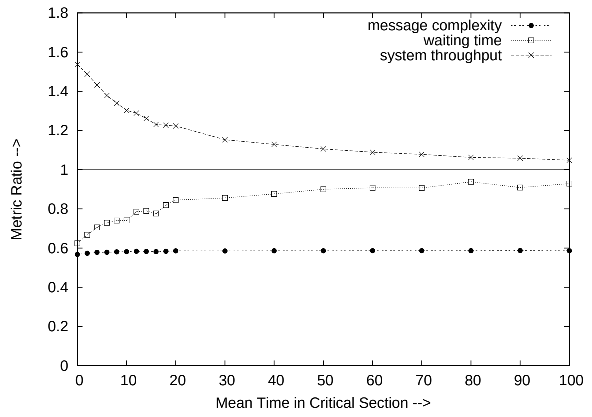
Our experimental study has the following parameters. There are n processes requesting entry into m different forums. A process, on generating a request, randomly selects a forum to join. The inter-request delay (that is, duration of non-critical section) at each process is exponentially distributed with mean μ_{ncs}. Once a process enters a forum, it departs after a delay that is uniformly distributed in the range $[0, 2 * \mu_{cs}]$ (duration of critical section). Message transmission delay (or channel delay) is modeled to follow an exponential distribution with mean μ_{cd}. In our experiments, parameters that have fixed values throughout are number of processes n, which is set to 25, channel bandwidth, which is set to 1000 integers per unit time, and number of requests per process, which is set to 1000. All other parameters are varied one by one to study their effect on the relative performance of the two algorithms.



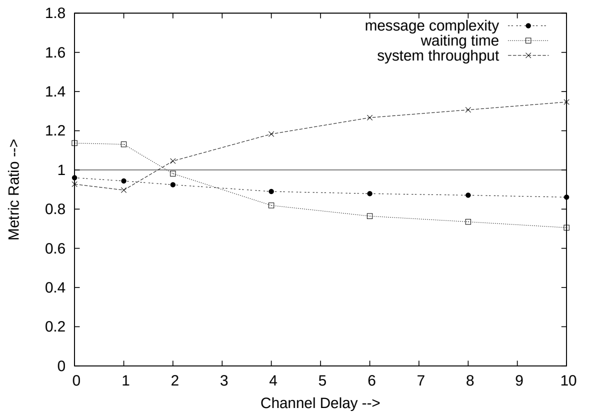
(a) Varying the number of forums (m) with $\mu_{ncs} = 4$ time units, $\mu_{cs} = 2$ time units and $\mu_{cd} = 4$ time units.



(b) Varying the mean inter-request delay (μ_{ncs}) with $m = 20$, $\mu_{cs} = 2$ time units and $\mu_{cd} = 4$ time units.



(c) Varying the mean time in critical section (μ_{cs}) with $m = 100$, $\mu_{ncs} = 4$ time units and $\mu_{cd} = 4$ time units.



(d) Varying the mean channel delay (μ_{cd}) with $m = 20$, $\mu_{ncs} = 4$ time units and $\mu_{cs} = 2$ time units.

Fig. 5. Relative performance of the two algorithms as a function of various parameters.

For Maekawa_M, the maximum number of simultaneous locks that a quorum node can grant is set to the maximum, that is, n. Also, to construct the surficial quorum system, $\sqrt{\frac{2n}{m(m-1)}}$ has to be an integer. If not, we add processes until $\sqrt{\frac{2\bar{n}}{m(m-1)}}$ becomes an integer, where $\bar{n}$ is the new value for n. The new processes are mapped onto existing processes in a round-robin fashion.

A. Simulation Results

Figure 5 depicts the variation in the ratios for the three metrics as a function of various parameters. The ratios are averaged over several runs to obtain 95% confidence level.

1) Variation with the number of forums: We vary the number of forums from 2 to 100 with the values for other parameters as follows: $\mu_{ncs} = 4$ time units, $\mu_{cs} = 2$ time units and $\mu_{cd} = 4$

time units. Figure 5(a) shows the variation in the ratios for the three metrics. When the number of forums is small (≤ 15), Maekawa_M has better message-complexity than Surrogate. However, in almost all the cases, Surrogate has better waiting time and system throughput than Maekawa_M. Further, for larger values for number of forums, Surrogate outperforms Maekawa_M with respect to all three metrics. Specifically, when the number of forums is large (≥ 90), message complexity and waiting time decrease by around 35% and system throughput increases by 48%.

2) Variation with the mean inter-request delay: We vary the mean inter-request delay from 0 to 100 time units with the values for other parameters as follows: $m = 20$, $\mu_{cs} = 2$ time units and $\mu_{cd} = 4$ time units. The results of our experiments are shown in Figure 5(b). As shown in the figure, Surrogate consistently outperforms Maekawa_M in all three complexity measures. However, the improvements in message complexity, waiting time and system throughput remain relatively constant at around 10%, 20% and 15%, respectively.

3) Variation with the mean time in critical section: We vary the mean time in critical section from 0 to 100 time units with the values for other parameters as follows: $m = 100$, $\mu_{ncs} = 4$ time units and $\mu_{cd} = 4$ time units. The variation in the three metric ratios is shown in Figure 5(c). The message complexity for Surrogate is 40% better than that for Maekawa_M. Further, our experiments demonstrate that, although Surrogate has better waiting time and system throughput than Maekawa_M, the gap in the performance decreases as the mean time in critical section increases. Nevertheless, if a process stays in its critical section for a relatively short amount of time, then Surrogate provides almost 35% improvement in waiting time and 45% improvement in system throughout.

4) Variation with the mean channel delay: We vary the mean channel delay from 0 to 10 time units with the values for other parameters as follows: $m = 20$, $\mu_{ncs} = 4$ time units and $\mu_{cs} = 2$ time units. Figure 5(d) shows the variation in the three metric ratios. For small values of mean channel delay, Maekawa_M has better performance than Surrogate. However, as the mean channel delay increases, the performance of Surrogate relative to Maekawa_M improves as well. Specifically, at larger values of mean channel delay, message complexity decreases by 20%, waiting time decreases by 30% and system throughput increases by 34%.

B. Discussion of Results

As our simulation results demonstrate, when the number of forums (m) or the mean channel delay (μ_{cd}) is small, Maekawa_M has somewhat better performance than Surrogate. In almost all other cases, Surrogate has better performance than Maekawa_M. Specifically, message complexity and waiting time decreases by as much as 40% and system throughput increases by as much as 48%. One of the reasons for the better performance of Surrogate may be the addition of new (logical) processes so that the surficial quorum system can be constructed. However, even when Maekawa_M is used with a grid quorum system (Toyomura *et al*'s algorithm), our simulation results show that Surrogate continues to have much better performance.

VIII. Discussion

In this section, we discuss several extensions to Surrogate for improving its performance.

A. Achieving Concurrent Entry

Surrogate does not satisfy the concurrent entry property. Once a requesting process enters a forum as a leader, any request generated thereafter has to wait until the current forum is dissolved even if all requests are of the same type. However, we can achieve concurrent entry by making the following modifications. Whenever a locked (quorum) node receives a request that is compatible with its locking request, it simply forwards that request to the locking process (via a FORWARD message) unless it is aware of a conflicting request that has not yet been fulfilled. A leader, on receiving a forwarded request, sends an INVITE message to the requesting process. In case a process receives a forwarded request before it has successfully locked all its quorum members, the request is stored along with other requests that it has received (or is going to receive) with LOCKED messages.

This forwarding of requests by a quorum node to the locking process continues until the node learns about a pending conflicting request. Specifically, a quorum node, on receiving a request that conflicts with that of the locking process, sends a STEPDOWN message to the process. If, at the time of sending a LOCKED message to a request, the node is aware of a conflicing request, then the STEPDOWN message is piggybacked on the LOCKED message itself. The purpose of the STEPDOWN message is to instruct the process to stop inviting followers into the forum. Otherwise, a conflicting request may get starved. This is because some other quorum node may continue to forward compatible requests to the process due to which the current forum may never dissolve. A leader stops inviting followers into the forum after either (1) receiving a STEPDOWN message from some quorum member sent on behalf of a conflicting request that has not yet been fulfilled or (2) leaving the critical section without inviting any follower (that is, its weight is one).

Clearly, with the above modifications, our algorithm satisfies the concurrent entry property. If all requests in the system are of the same type as the current forum and no conflicting request is ever generated, then each request is fulfilled within three message hops. Starvation freedom is guaranteed because a leader stops sending INVITE messages after it has received any STEPDOWN message. Once a conflicting request is generated, leader of the current forum receives a STEPDOWN message within two message hops. In other words, a leader can send only a finite number of INVITE messages once a conflicting request is generated in the system.

As regards to the message complexity, note that a FORWARD or a STEPDOWN message is generated only when a new request arrives at a quorum node and never for an old request already present in the queue. Therefore at most one FORWARD or STEPDOWN message is sent by a quorum member for every request it receives. Therefore the message complexity remains at $O(q)$. The synchronization delay remains at two message hops because, at heavy loads, STEPDOWN message is piggybacked on the LOCKED message itself. The waiting time also does not increase. At light loads, a process entering the forum as a leader does not have any followers and therefore releases its locks on quorum members soon after leaving the critical section. Our experimental results show that the modification for concurrent entry results in only a small improvement in the performance.

B. Achieving Unnecessary Blocking Freedom

Consider the CD jukebox example. Suppose some data is replicated on multiple CDs. Therefore any request for such data can be satisfied using any *one* of the CDs on which data has been replicated. In traditional group mutual exclusion, a process has to specify the type of critical section it wants to execute at the time of the request—which translates into the CD it wants to access for satisfying its request. This may lead to *unnecessary delay (or blocking)* in satisfying a request. To eliminate this unnecessary delay, Manabe and Park extend the group mutual exclusion problem in [22] to allow a process to specify more than one type when making a request. The request can be satisfied by allowing a process to execute critical section for any one of those types (specified at the time of the request).

Surrogate can be easily modified to achieve unnecessary blocking freedom as follows. For a request α, let $\textit{type-set}(\alpha)$ denote the set of types that can be used to satisfy α. A quorum member, on sending a LOCKED message to a request α, piggybacks all requests β on the LOCKED message for which $\textit{type-set}(\alpha) \cap \textit{type-set}(\beta) \neq \emptyset$. As before, once a request has successfully locked all its quorum members, it enters the forum as a leader. However, at the time of entry, it still has to select a type for the forum (which translates into selecting a CD to be loaded into the jukebox).

For a request α satisfied as a leader, let *intersecting-set*(α) denote the set of requests it has received from its quorum members along with LOCKED messages. Observe that it is possible for *intersecting-set*(α) to contain two requests β and γ such that: (1) *type-set*$(\alpha) \cap$ *type-set*$(\beta) \neq \emptyset$, (2) *type-set*$(\alpha) \cap$ *type-set*$(\gamma) \neq \emptyset$, and (3) *type-set*$(\beta) \cap$ *type-set*$(\gamma) = \emptyset$. Clearly, α cannot send INVITE messages to both β and γ. To maximize concurrency, α chooses a type $t \in$ *type-set*(α) for which the number of requests in *intersecting-set*(α) that are "compatible" with t is *maximized*. All properties of Surrogate are preserved except for message overhead which increases by a factor of s, where s is the maximum number of types a process can specify at the time of request.

C. A Generalized Algorithm for Group Mutual Exclusion using Information Structure

Sanders presents a generalized mutual exclusion algorithm in [23] that unifies several (permission-based) mutual exclusion algorithms using the concept of *information structure*. Specifically, each process x has two sets associated with it, namely *request set* and *inform set*, denoted by R_x and I_x, respectively, with $x \in I_x$ and $I_x \subseteq R_x$. Mutual exclusion is guaranteed under the following condition: for all processes x and y, either $I_x \cap I_y \neq \emptyset$ or $x \in R_y \wedge y \in R_x$ holds. By choosing request and inform sets appropriately, it is possible to derive both Ricart and Agrawala's algorithm [4] and Maekawa's algorithm [19] as special cases of the generalized algorithm.

The main idea behind the generalized algorithm is as follows. A process sends its request to all processes in its request set. It can enter the critical section once it has received a GRANT message (similar to a LOCKED message) from all nodes in its request set. However, on leaving the critical section, it sends a RELEASED message to nodes in its inform set only. This is because only nodes in its inform set "remember" sending a GRANT message to it (and are waiting to receive a RELEASED message from it before they can send another GRANT message). Other nodes in its request set, which are not in its inform set, simply send a GRANT message to it without maintaining any state information about the process. This, in turn, implies that a node can continue to send a GRANT message until it sends a GRANT message to a process such that it belongs to the inform set of that process. Deadlocks are avoided in a similar fashion as in Maekawa's algorithm by using FAILED, INQUIRE and RELINQUISH messages.

The ideas developed in this paper can be combined with the information structure proposed by Sanders to derive a generalized algorithm for group mutual exclusion. A node, on sending a GRANT message to a process, piggybacks on the message the list of processes that have pending requests compatible with that of the process to which the GRANT message is being sent. Once a process has received GRANT messages from all nodes in its request set, it enters the forum as a leader. It then sends INVITE messages to all processes with pending requests that are compatible with its own request. These processes enter the forum as followers. As in Surrogate, a list of latest stale requests can be maintained to avoid generating stale INVITE messages. Further, weight distribution and recovery scheme [24] can be employed to enable a follower to send RELEASED messages directly to nodes in the leader's inform set. The proof of correctness is very similar to that of Surrogate.

Let r denote the maximum size of a request set. It can be verified that the worst-case message complexity of the generalized algorithm is given by $O(r)$. Further, let s denote the maximum number of request sets to which a node can belong. The amortized message overhead of the generalized algorithm is given by $O(s)$. Clearly, its (best-case) synchronization delay and waiting time are both two message hops.

1) Modification when Inform Sets are Singleton: Note that, in the information structure for Ricart and Agrawala's algorithm [4], an inform set of a process only contains itself (inform set is singleton). In the generalized mutual exclusion algorithm by Sanders, a process sends RELEASED messages only to nodes in its inform set. As a result, only a GRANT message contributes towards the synchronization delay (a RELEASED message does not incur any transmission delay).

When Ricart and Agrawala's information structure is used with the generalized group mutual exclusion algorithm, however, the synchronization delay increases to two message hops. This is because a node may now need to wait to receive a RELEASED message from a process that entered the forum as a follower. Clearly, unlike the RELEASED message of a leader, the RELEASED message of a follower may incur non-negligible transmission delay. To reduce the synchronization delay, we make the following modification. A node does not need to wait to receive RELEASED messages from followers; it resumes sending GRANT messages as soon as it has received a RELEASED message from itself. A process receiving the GRANT message still has to wait until it has received RELEASED messages from all processes that entered the current forum as followers before starting the next forum. A follower, on leaving the forum, sends a RELEASED message directly to all processes to whose REQUEST set its leader belongs. This increases the message complexity to $O(r+s)$, which is $O(n)$ because, with Ricart and Agrawala's information structure, $r = s = n$.

IX. Conclusion

We have proposed an efficient distributed algorithm for solving the group mutual exclusion problem based on the notion of *surrogate-quorum*. Unlike the existing quorum-based algorithms for group mutual exclusion [20], our algorithm achieves a low message complexity of $O(q)$ while at the same time maintaining both synchronization delay and waiting time at two message hops. In doing so, we introduce a relatively low message overhead of $O(b)$ amortized over all messages. If Maekawa's grid quorum system [19] is used, then the message complexity and the amortized message overhead are both given by $O(\sqrt{n})$. Our algorithm has high maximum concurrency of n. Furthermore, unlike the algorithms in [20], which assume that the number of groups is static and does not change during runtime, our algorithm can adapt without performance penalties to dynamic changes in the number of groups. Our experimental results indicate that Surrogate has better performance than Maekawa_M—another quorum-based group mutual exclusion algorithm—in almost cases except when the number of forums or the mean channel delay is small.

We have also presented several extensions to our basic algorithm for achieving other desirable properties. Furthermore, we have provided a generalized algorithm for group mutual exclusion by combining the concept of surrogate-quorum with that of information structure.

A natural extension of group mutual exclusion is to allow up to k forums to be in session simultaneously, where $k \geq 1$. This corresponds to the scenario when the CD jukebox has more than player and therefore multiple CDs can be loaded simultaneously. As a future work, we plan to devise an efficient distributed algorithm for solving this problem.

References

[1] R. Atreya and N. Mittal, "A Dynamic Group Mutual Exclusion Algorithm using Surrogate-Quorums," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, Columbus, Ohio, USA, June 2005, pp. 251–260.

[2] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM (CACM)*, vol. 8, no. 9, p. 569, 1965.

[3] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, July 1978.

[4] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM (CACM)*, vol. 24, no. 1, pp. 9–17, Jan. 1981.

[5] I. Suzuki and T. Kasami, "A Distributed Mutual Exclusion Algorithm," *ACM Transactions on Computer Systems*, vol. 3, no. 4, pp. 344–349, 1985.

[6] K. Raymond, "A Tree based Algorithm for Distributed Mutual Exclusion," *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 61–77, 1989.

[7] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource Allocation with Immunity to Limited Process Failure (preliminary report)," in *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, Oct. 1979, pp. 234–254.

[8] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, Oct. 1971.

[9] K. M. Chandy and J. Misra, "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 4, pp. 632–646, 1984.

[10] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[11] Y.-J. Joung, "Asynchronous Group Mutual Exclusion," *Distributed Computing (DC)*, vol. 13, no. 4, pp. 189–206, 2000.

[12] P. Keane and M. Moir, "A Simple Local-Spin Group Mutual Exclusion Algorithm," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 1999, pp. 23–32.

[13] K. Alagarsamy and K.Vidyasankar, "Elegant Solutions for Group Mutual Exclusion Problem," Department of Computer Science, Memorial University of Newfoundland, St.John's, Newfoundland, Canada, Tech. Rep., 1999.

[14] V. Hadzilacos, "A Note on Group Mutual Exclusion," in *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 2001.

[15] S. Cantarell, A. K. Datta, F. Petit, and V. Villain, "Token Based Group Mutual Exclusion for Asynchronous Rings," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2001, pp. 691–694.

[16] K.-P. Wu and Y.-J. Joung, "Asynchronous Group Mutual Exclusion in Ring Networks," *IEEE Proceedings—Computers and Digital Techniques*, vol. 147, no. 1, pp. 1–8, 2000.

[17] J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit, "Group Mutual Exclusion in Tree Networks," *Journal of Information Science and Engineering*, vol. 19, no. 3, pp. 415–432, May 2003.

[18] Y.-J. Joung, "The Congenial Talking Philosophers Problem in Computer Networks," *Distributed Computing (DC)*, pp. 155–175, 2002.

[19] M. Maekawa, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, May 1985.

[20] Y.-J. Joung, "Quorum-Based Algorithms for Group Mutual Exclusion," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 5, pp. 463–475, May 2003.

[21] M. Toyomura, S. Kamei, and H. Kakugawa, "A Quorum-Based Distributed Algorithm for Group Mutual Exclusion," in *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Chengdu, Sichuan, China, Aug. 2003, pp. 742–746.

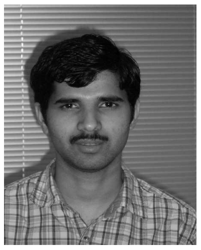
[22] Y. Manabe and J. Park, "A Quorum-Based Extended Group Mutual Exclusion Algorithm without Unnecessary Blocking," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, Newport Beach, California, USA, 2004, pp. 341–348.

[23] B. A. Sanders, "The Information Structure of Distributed Mutual Exclusion Algorithms," *ACM Transactions on Computer Systems*, vol. 5, no. 3, Aug. 1987.

[24] F. Mattern, "Global Quiescence Detection based on Credit Distribution and Recovery," *Information Processing Letters (IPL)*, vol. 30, no. 4, pp. 195–200, 1989.

Ranganath Atreya received his Bachelor's degree with distinction from Bangalore University, India in 2000 and his M.S. degree in computer science from The University of Texas at Dallas in 2004. He was the recipient of the National Award for the best Bachelor of Engineering project in the year 2000. He has been working at Amazon.com, Inc. since January 2005.

Neeraj Mittal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. degrees in computer science from The University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a co-director of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking and databases.



Sathya Peri received his M.C.S.A degree in computer science from Madurai Kamaraj University, India in 2001. He worked as a software engineer at HCL Technologies, India for one year from 2001 to 2002. He is currently pursuing his Ph.D. degree in computer science at Advanced Networking and Dependable Systems Laboratory (ANDES) at The University of Texas at Dallas. His research interests include peer-to-peer computing and dynamic distributed systems.