

A Dynamic Group Mutual Exclusion Algorithm using Surrogate-Quorums

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

Neeraj Mittal
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083, USA
neerajm@utdallas.edu

Abstract

The group mutual exclusion problem extends the traditional mutual exclusion problem by associating a type 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. In this paper, we provide 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, like the existing quorum-based algorithms, our algorithm has high maximum concurrency of n , where n is the number of processes in the system. The existing quorum-based algorithms assume that the number of groups is static and does not change during runtime. However, our algorithm can adapt without performance penalties to dynamic changes in the number of groups. Simulation results indicate that our algorithm outperforms the existing quorum-based algorithms for group mutual exclusion by as much as 50% in some cases.

1. Introduction

Mutual exclusion is one of the most fundamental problems in concurrent systems, particularly in distributed systems. In this problem, access to a shared resource (that is,

*This work was done while the author was a student in the Department of Computer Science at The University of Texas at Dallas

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 [6, 15, 19, 20, 18] and extensions [8, 7, 5, 13] 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 [10]. 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 in CDs and only one disk can be loaded for access at a time [10]. 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 a different disk will have to wait.

Solutions for the group mutual exclusion problem under shared-memory model can be found in [10, 14, 1, 9]. 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 [4, 21] and tree networks [3]. 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 [19] have been proposed in [11]. These algorithms have high message complexity of $O(n)$. Further, the first algorithm has low expected concurrency of $O(1)$, whereas the second algorithm has high message overhead of $O(n)$.

The quorum-based mutual exclusion algorithm by Maekawa [16] has also been modified to derive two quorum-based algorithms for group mutual exclusion [12]. These algorithms use a special type of quorum system called the 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. The maximum number of pair-wise disjoint quorums offered by a group quorum system is called the *degree* of the quorum system. In [12], 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 an immutable number of groups.

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.

In this paper, we take an approach that is orthogonal to existing approaches for quorum-based algorithms and introduce the notion of *surrogate-quorums*. The existing quorum-based algorithms [12] 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 [12] 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 present an extension to our algorithm for achieving *concurrent entry* without incurring any performance penalties.

The rest of the paper is organized as follows. We present our system model and formally describe the group mutual exclusion problem in Section 2. We provide the necessary background information in Section 3. We then present our surrogate-quorum based algorithm for group mutual exclusion in Section 4. We prove its correctness in Section 5 and analyze its performance in Section 6. An extension to our algorithm for achieving concurrent entry is described in Section 7. We present our simulation results in Section 8 and our conclusions in Section 9.

2. Model and Problem Definition

2.1. System Model

We assume an asynchronous distributed system comprising of multiple processes which 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.

2.2. The Group Mutual Exclusion Problem

The problem of *group mutual exclusion (GME)* was first proposed in [10] 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*. Process can make requests for arbitrary and unrestricted types of critical sections. 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 will eventually succeed.

concurrent entry: 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.

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 all processes executing critical sections of the same type exit (their critical sections) and when another process can enter a critical section of a different type
- *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
- *concurrency:* the number of processes that are in their critical sections at the same time

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. Waiting time is typically measured when the system is lightly loaded.

3. Background

3.1. Quorum System

A *quorum* is a subset of nodes or processes. Although nodes and processes are identical, following the convention in [12], 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 , where n is the number of processes. 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:

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

Existing quorum-based algorithms [12] 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.

3.2. Maekawa’s Algorithm

Maekawa’s algorithm implements mutual exclusion by using a coterie that satisfies the aforementioned properties. Lamport’s logical clock [15] 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 exiting 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 [16] 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 [16, 12].)

4. Surrogate-Quorum based Algorithm

We now describe our approach for solving the group mutual exclusion problem. For convenience, we say that the first process to enter a critical section of some type *initiates* a *forum* of that type. Further, the last process to exit the critical section of that type terminates the forum. Intuitively, a forum contains one of more critical sections of the same type. We call two requests as *compatible* if they are for the same forum type; otherwise they are said to be *conflicting*.

4.1. 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 [12], 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 3.1.

One approach to achieving concurrency is by enabling nodes to issue multiple locks [12]. 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 [11] 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.

4.2. Reducing Synchronization Delay

Since synchronization delay has a significant impact on efficiency, especially system throughput, 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 [17]. 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 $O(b)$.

4.3. 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 next leader, due to the intersection property, has to obtain LOCKED message from at least one quorum member of the previous leader. Hence there exists a causal path from all processes that entered some forum, to the leader of the succeeding forum. In this modification, we basically exploit this causal path to pass on information about stale requests to the next leader. We propose the following modifications.

1. Each process x maintains a vector of timestamps; there is one entry in the vector for each process in the system. The entry for process y in the vector contains the timestamp of the latest request by process y that has been fulfilled according to x 's knowledge.
2. A node, on receiving a RELEASED message from a process updates its vector with the timestamp of the request that sent the RELEASED message. If there already exists 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 vector that have not been previously sent to the process.
4. 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, 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 $O(n)$, where n denotes the number of processes. But as we shall show later, this overhead in fact amortizes to $O(b)$ over all messages.

We refer to our algorithm as **Surrogate**. A more formal description of **Surrogate** can be found in [2].

5. 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 7, we give a simple extension to **Surrogate** and show that it achieves concurrent entry.

We now explore the behavior of the system. The period when no process is in the forum is the period when there is no leader in the system and hence called *anarchy*. The system starts in anarchy and processes with outstanding requests compete to capture their respective quorums. A request 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 process. Wait-for-subgraphs may branch out and form more wait-for-subgraphs. However as we shall show later, these wait-for-subgraphs eventually unlink. When a process succeeds in capturing all its quorum members, it enters the forum as a leader. A leader sends out **INVITE** messages to processes requesting entry into the leader's forum as and when the leader becomes aware of their requests and these processes enter the forum as followers. The forum thus formed remains in *session* as long as there is at least one process in it. When all processes exit the forum, it is dissolved and the system enters anarchy again. The cycle continues until all requests are fulfilled. We now formally prove that the system in fact behaves as described.

In the following proofs, we model the execution of the system as an infinite alternating sequence of global states and events, $\sigma = S_0 e_1 S_1 \cdots S_{i-1} e_i S_i \cdots$. For executions with finite number of such global states, we assume the existence of a hypothetical *nop* event for no-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 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 . A continuously enabled event is assumed to be eventually executed. We use the notation $MSG^i(x, y)$ to mean that at state S_i , x sent y the message MSG .

5.1. Group Mutual Exclusion

The group mutual exclusion property dictates that the algorithm allow at most one forum to be in session at any given instant in time. Stated another way, no two processes can be in different forums at the same time. To prove the safety property, we use certain invariants of the algorithm which we state without proof as they are evident from the algorithm. Henceforth, a request is said to be *extant* at a state if the request is either outstanding or executing in the given state. Also, a

request is said to be *stale* in all states following its entry into the forum. The following notation is used in the proofs.

$$\begin{aligned} request^i(x) &\triangleq \text{process } x \text{ has an extant request at state } S_i \\ forum^i(x) &\triangleq x \text{ is in some forum at global state } S_i \\ quorum^i(x) &\triangleq \text{the quorum chosen by the extant request of process } x \text{ at } S_i \text{ (defined only if } request^i(x)) \end{aligned}$$

Invariant 1 For every process in some forum, there exists at least one quorum that has not released its locks.

Invariant 2 A node does not grant a **LOCKED** message to a request until all processes have left the current forum and sent it **RELEASED** messages.

Theorem 1 **Surrogate** satisfies safety property, that is, no two forums of different types execute concurrently.

Proof: From Invariant 1, for every forum there exists at least one quorum, say Q , that has not released its locks. From intersection property, every process that has requested a different forum must acquire locks from at least one node in Q , before entering its forum. From Invariant 2, no node in Q will release its lock until all processes have exited the current forum. Hence, two forums of different types cannot execute concurrently. \square

5.2. Starvation Freedom

Starvation freedom property for group mutual exclusion dictates that every request made in the system be eventually fulfilled. In this section we prove that **Surrogate** satisfies starvation freedom property. In addition, we use the notations described in Figure 1.

In order to prove starvation freedom, we first formalize the concept of wait relationships and wait-for-graphs. We say that a process x is *waiting* on another process y at global state S_i if process x is waiting for a lock that was sent to y and $ts^i(x) < ts^i(y)$ (ties are broken using process identifiers). Formally,

$$\begin{aligned} wait^i(x, y) &\triangleq \\ &\langle \exists z : z \in quorum^i(x) \cap quorum^i(y) : \\ &\quad x \in queue^i(z) \wedge locked^i(z, y) \rangle \wedge \\ &(ts^i(x) < ts^i(y)) \end{aligned}$$

Based on the above definition of wait relation, each state of a distributed system running **Surrogate** can be represented by a directed graph. In this graph for a given state, the set of vertices is the set of processes in the system and a directed edge exists from vertex u to v if process u is waiting on process v in the given state. Such a graph for a given

P	\triangleq	set of all processes in the system
$ts^i(x)$	\triangleq	the timestamp of the extant request of process x at state S_i (defined only if $request^i(x)$ holds).
$locked^i(x, y)$	\triangleq	At state S_i the following holds: node x has sent its lock to process y for a request that is still extant but y has not yet relinquished that lock nor cancelled the request it sent to x .
$queue^i(x)$	\triangleq	the set of processes that belong to request queue of process x at state S_i
$failcount^i(x)$	\triangleq	number of FAILED messages sent to request of process x at S_i , from the state in which it was issued till state S_i (defined only if $request^i(x)$)

Figure 1. Notation used in proofs

state S_i is called the *wait-for-graph* at state S_i denoted by, $WFG^i(P)$.

Since every outstanding request belongs to the wait-for-graph, we can define *wait-for-subgraph* emanating from a given process at some state S_i as the subgraph of the wait-for-graph at S_i in which every vertex is reachable from the emanating vertex. The wait-for-subgraph emanating from process x at state S_i is denoted by $WFG^i(x)$.

In addition, we use $wait-set^i(x)$ to indicate the set of all processes that x is directly waiting on at state S_i . We shall now define certain invariants and properties of the algorithm without proof as they directly follow from the algorithm.

Invariant 3 Every request eventually receives a FAILED message, enters the forum or waits on another request. Formally,

$$request^i(x) \Rightarrow \langle \exists j : j \geq i : failcount^j(x) > 0 \vee \\ wait-set^j(x) \neq \phi \vee \\ forum^j(x) \rangle$$

Property A $WFG^i(P)$ is acyclic. Since the time stamp along any path monotonically increases.

Property B As a consequence, all paths in $WFG^i(P)$ are simple paths.

Property C The maximum length of all paths in $WFG^i(P)$ is bounded by $|P|$.

Due to space constrains, we only provide statements of the lemmas here and refer the reader to [2] for the omitted proofs.

Lemma 2 Surrogate does not generate stale INVITE messages.

Lemma 3 Once a leader exits the forum, eventually all its quorum members recover their locks.

It remains to be shown that deadlock resolution through INQUIRE, RELINQUISH and FAILED messages do in fact avoid deadlocks. In this regard, we explore the behavior of nodes in a wait-set.

Lemma 4 For any node y in the wait-set of some node x , if y enters the forum or receives a FAILED message then either x eventually receives a FAILED message or y eventually leaves the wait-set of x . Formally,

$$y \in wait-set^i(x) \wedge (forum^i(y) \vee failcount^i(y) > 0) \\ \Rightarrow \\ \langle \exists j : j \geq i : y \notin wait-set^j(x) \vee failcount^j(x) > 0 \rangle$$

Lemma 5 Every request is eventually fulfilled or its fail count becomes non-zero. Formally,

$$request^i(x) \Rightarrow \langle \exists j : j \geq i : forum^j(x) \vee \\ failcount^j(x) > 0 \rangle$$

In order to prove starvation freedom, we shall now formally define two attributes of a request: *potence* and *omnipotence* denoted by $potent^i(x)$ and $omnipotent^i(x)$ respectively. A request is $potent^i(x)$ if and only if no request of higher priority is ever generated in all states following S_i . Further, a request is $omnipotent^i(x)$ if and only if it is $potent^i(x)$ and has the highest priority among all requests in S_i .

Lemma 6 Every omnipotent request is eventually fulfilled. Formally,

$$omnipotent^i(x) \Rightarrow \langle \exists j : j \geq i : forum^j(x) \rangle$$

In the following two lemmas we show that every request eventually becomes an omnipotent request or gets fulfilled. Hence from Lemma 6 every request will be eventually fulfilled.

Lemma 7 *Every request eventually becomes a potent request or is fulfilled. Formally,*

$$request^i(x) \Rightarrow \langle \exists j : j \geq i : potent^j(x) \vee forum^j(x) \rangle$$

Lemma 8 *Every potent request eventually becomes the omnipotent request or gets fulfilled. Formally,*

$$potent^i(x) \Rightarrow \langle \exists j : j \geq i : forum^j(x) \vee omnipotent^j(x) \rangle$$

Theorem 9 *Every request is eventually fulfilled. Formally,*

$$request^i(x) \Rightarrow \langle \exists j : j \geq i : forum^j(x) \rangle$$

Proof: The theorem directly follows from Lemma 7, Lemma 8 and Lemma 6. \square

6. Performance 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 10 *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 3, 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. \square

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 11 *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 $O(b)$. \square

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 12 *The synchronization delay and the waiting time for Surrogate are both two message hops.*

As before n denotes the number of processes in the system. Clearly, if all processes make compatible requests, then all of them can be in the forum at the same time.

Theorem 13 *The maximum concurrency of Surrogate is n*

7. Discussion

The algorithm Surrogate does not satisfy the concurrent entry property. However, by making the following simple

modification we can achieve concurrent entry. If a node is locked by a process and receives a request that is compatible with its locking request, then it simply forwards that request to the locking process. A leader on receiving a forwarded request sends `INVITE` to the requesting process. In order to prevent starvation, a node asks a leader to stop sending `INVITE` to forwarded requests, as and when it becomes aware of a conflicting request. Clearly, with the above modifications, our algorithm satisfies the concurrent entry property. Note that an additional message is generated only when a new request arrives at a quorum node and never for an old request that is already in the queue. Therefore the message complexity remains at $O(q)$. Also, the synchronization delay and the maximum concurrency remain at two message hops and n , respectively. The reader is referred to [2] for details.

The concurrency in our algorithm can be further enhanced by increasing the set of processes that “know” about a request. This is because, a process is only invited by a leader when its request is piggybacked over at least one of the `LOCKED` messages sent to the leader. If the probability of a request being piggybacked increases, then the probability of the request being invited also increases, thereby increasing concurrency. To that end, we introduce the notion of a *notify set*. The notify set for a process is a set of nodes that intersects with all quorum sets other than its own quorum set. A process upon making a request, only informs the nodes in its notify set and does not try to lock them. Intuitively, it increases the number of nodes that are aware of a particular request, which, in turn, increases the probability of that request being invited without increasing the waiting time.

8. Simulation Results

In this section, we experimentally compare the performance of `Surrogate` (without modification for concurrent entry) with Joung’s first algorithm `Maekawa_M`. In the simulation, we have n processes requesting entry into m forums. Each process makes a request to enter a randomly chosen forum, uniformly distributed over all m forums. The inter-request delay at each node is exponentially distributed with mean μ_{idle} . Once a process enters a forum, it departs after a delay that is uniformly distributed in the range $[0, 2 * \mu_{forum}]$. Message transmission delay is modeled to follow an exponential distribution with mean μ_{link} .

We measure the performance of the two algorithms with respect to three metrics, namely message complexity, system throughput and waiting time. All simulations are conducted with the following values for various parameters: $n = 25$, average requests per node = 1000, $\mu_{forum} = 2$ time units, $\mu_{idle} = 4$ time units, $\mu_{link} = 4$ time units, and bandwidth = 1000 integers per unit time. For `Maekawa_M`, the maximum number of simultaneous locks that a node can

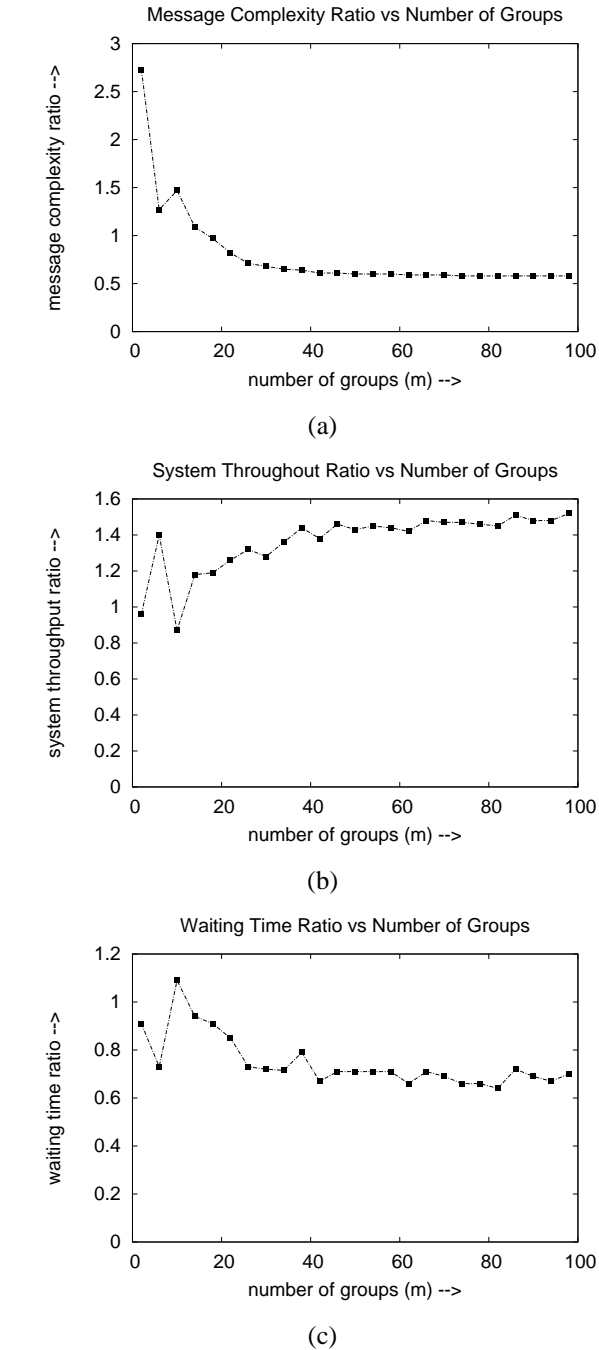


Figure 2. Performance of Surrogate relative to Maekawa_M.

grant is set to n .

To make it easier to compare the two algorithms, we report the ratio `Surrogate/Maekawa_M` for the three metrics. Figure 2 depicts the variation in the ratios for the three metrics as the number of groups increases. The ra-

tios somewhat fluctuate for values of $m < n$. However, for all values of $m \geq n$, **Surrogate** performs consistently better than **Maekawa_M**. The message complexity decreases by as much as 40%, the system throughput increases by as much as 50% and the waiting time decreases by as much as 34%. For a more detailed experimental analysis, the reader is referred to [2].

9. Conclusion

In this paper, we describe 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 [12], our algorithm achieves a low message complexity of $O(q)$, where q is the maximum size of a quorum, 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, where b denotes the size of the largest membership set. If Maekawa's grid quorum system [16] 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 , where n is the number of processes in the system. Furthermore, unlike the algorithms in [12], 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. We also describe an extension to our algorithm for achieving concurrent entry.

References

- [1] K. Alagarsamy and K. Vidasankar. Elegant Solutions for Group Mutual Exclusion Problem. Technical report, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, 1999.
- [2] R. Atreya. A Dynamic Group Mutual Exclusion Algorithm using Surrogate-Quorums. Master's thesis, Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA, Dec. 2004.
- [3] J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit. Group Mutual Exclusion in Tree Networks. *Journal of Information Science and Engineering*, 19(3):415–432, May 2003.
- [4] 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)*, pages 691–694, 2001.
- [5] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.
- [6] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM (CACM)*, 8(9):569, 1965.
- [7] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, Oct. 1971.
- [8] 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)*, pages 234–254, Oct. 1979.
- [9] V. Hadzilacos. A Note on Group Mutual Exclusion. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 2001.
- [10] Y.-J. Joung. Asynchronous Group Mutual Exclusion. *Distributed Computing (DC)*, 13(4):189–206, 2000.
- [11] Y.-J. Joung. The Congenial Talking Philosophers Problem in Computer Networks. *Distributed Computing (DC)*, pages 155–175, 2002.
- [12] Y.-J. Joung. Quorum-Based Algorithms for Group Mutual Exclusion. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(5):463–475, May 2003.
- [13] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [14] P. Keane and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 1999.
- [15] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.
- [16] M. Maekawa. A \sqrt{N} Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.
- [17] F. Mattern. Global Quiescence Detection based on Credit Distribution and Recovery. *Information Processing Letters (IPL)*, 30(4):195–200, 1989.
- [18] K. Raymond. A Tree based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
- [19] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM (CACM)*, 24(1):9–17, Jan. 1981.
- [20] I. Suzuki and T. Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.
- [21] K.-P. Wu and Y.-J. Joung. Asynchronous Group Mutual Exclusion in Ring Networks. *IEEE Proceedings—Computers and Digital Techniques*, 147(1):1–8, 2000.