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

Ranganath Atreya, Neeraj Mittal, *Member*, *IEEE Computer Society*, and Sathya Peri

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 $O(q)$ and a low (amortized) bit-message complexity of $O(bqr)$, where q is the maximum size of a quorum, b is the maximum number of processes from which a node can receive critical section requests, and r is the maximum size of a request while maintaining both synchronization delay and waiting time at two message hops. 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 algorithm outperforms the existing quorum-based algorithms for group mutual exclusion by as much as 45 percent 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.

Index Terms—Message-passing system, resource management, mutual exclusion, group mutual exclusion, quorum-based algorithm.

1 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 (GME)*, has been proposed [11]. In the GME problem, every critical section is associated with a type or a group. Critical sections belonging to the same group can be executed concurrently, whereas 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 GME using $n+1$ groups, where n denotes the number of processes in the system. In this case, all *read* requests belong to the same group, and *write* requests by each process belong 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, whereas users that need data on different disks have to wait for the currently loaded disk to be unloaded.

Solutions for the GME problem have been proposed under both shared-memory and message-passing models. Solutions under the shared-memory model can be found in [11], [14], [15], [16], and [17]. In this paper, we investigate the GME problem under the message-passing model. For the message-passing model, solutions to GME have been proposed for ring networks [18], [19] and tree networks [20]. Typically, solutions for ring and tree networks incur high synchronization delay and have high waiting time. For a fully connected network, two GME algorithms based on the modification of the Ricart and Agrawala's algorithm for mutual exclusion [4] have been proposed in [21]. These algorithms have high message complexity of $O(n)$. The first algorithm has low expected concurrency of $O(1)$, whereas the second algorithm has high message and bit-message complexities of $O(n)$ and $O(n^2 r)$, respectively, where r is the maximum size of a request. (A request basically consists of three components: 1) the logical time stamp of the request, 2) the process that generated the request, and 3) the type of the request. Note that $r = \Omega(\log n + \log m)$, where m denotes the number of groups in the system.)

The quorum-based mutual exclusion algorithm of Maekawa [22] has also been modified to derive two quorum-based algorithms for GME [12]. 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, whereas quorums

• *The authors are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083.*
E-mail: ratreya@amazon.com, neerajm@utdallas.edu, sathya.p@student.utdallas.edu.

Manuscript received 30 Nov. 2005; revised 1 Sept. 2006; accepted 5 Dec. 2006; published online 26 Jan. 2007.
Recommended for acceptance by X. Jia.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0496-1105.
Digital Object Identifier no. 10.1109/TPDS.2007.1072.

TABLE 1
Comparison of Various Quorum-Based GME Algorithms

Algorithm	Complexity Measures				Assumptions
	Message Complexity	Bit-message Complexity	Synchronization Delay*	Waiting Time	
Maekawa_M [12]	$O\left(\frac{qn}{d}\right)$	$O\left(\frac{qnr}{d}\right)$	2	2	fixed set of groups
Maekawa_S [12]	$O(q)$	$O(qr)$	$q+1$	$q+2$	fixed set of groups
Toyomura *et al.* [13]	$O(nq)$	$O(nqr)$	2	2	-
Surrogate [this paper]	$O(q)$	$O(bqr)^{\dagger}$	2	2	-

$*$: ignoring delay due to deadlock avoidance
$\dagger$: amortized over all requests
n: number of processes in the system
q: maximum number of nodes in a quorum
d: degree of the group quorum system ($d \leq \sqrt{n}$)
r: maximum size of a request
b: maximum number of processes from which a node can receive requests for critical section

belonging to different groups must intersect. The maximum number of pairwise 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

$$\sqrt{\frac{2n}{m(m-1)}}.$$

When used with Maekawa's algorithm, the surficial quorum system can only allow up to the degree number of processes 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 Maekawa's original quorum-based algorithm. The first modification enables a node to issue multiple locks to requests belonging to the same group. The drawback of this approach is that in the event of conflict between requests of different groups, deadlock avoidance requires multiple locks to be taken back. This results in a high (worst case) message complexity of $O(\frac{qn}{d})$, where d is the degree of the underlying group quorum system (it can be proved that $d \leq \sqrt{n}$ [12]). To overcome this drawback, Joung proposed a second modification, which avoids deadlocks altogether. Specifically, deadlocks are avoided by locking quorum members in some fixed order. Although this approach reduces the message complexity to $O(q)$, the synchronization delay evidently increases from 2 to $O(q)$ message hops, where q is the maximum size of a quorum. In addition, both of these algorithms need an a priori knowledge of the number of groups to construct the group quorum system, 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 GME 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 [13]. Their algorithm is similar to Maekawa_M and therefore has a 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 [12] can provide either a low synchronization delay of two message hops or a low message complexity of $O(q)$, but not both. Hence, they are either inefficient or nonscalable. 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 exchange $O(bqr)$ bits per request (amortized over all requests), where b denotes the maximum number of processes from which a node can receive requests for critical section, and r denotes the maximum size of a request. As a result, our algorithm has an amortized bit-message complexity of $O(bqr)$.

Some of the existing quorum-based algorithms, such as those in [12], use a group quorum system whose construction may require an a priori knowledge of the number of groups in the system. Our algorithm uses a traditional quorum system, which does not depend on the number of groups in the system. Consequently, our algorithm can adapt without performance penalties to dynamic changes (at runtime) in the number of groups. Finally, as with the other quorum-based algorithms, it is possible for all processes to execute their critical sections concurrently if all of them request critical sections belonging to the same group. Table 1 compares the performance of various quorum-based GME algorithms. When our algorithm is used with the grid quorum system, for which $q = b = 2\sqrt{n} - 1$, it clearly outperforms the other three quorum-based algorithms. Later in this paper, we will also describe how our algorithm can be extended to satisfy certain desirable properties such as *concurrent entry* [21] and *unnecessary blocking freedom* [23].

The rest of the paper is organized as follows: We present our system model and formally describe the GME problem in Section 2. Section 3 describes the background necessary to understand our algorithm. We then present our quorum-based algorithm for GME in Section 4. We prove its correctness in Section 5, analyze its performance in Section 6, and present our simulation results in Section 7. In Section 8, we discuss several extensions to our basic algorithm. Finally, we present our conclusions in Section 9.

2 Model and Problem Definition

2.1 System Model

We assume an asynchronous distributed system comprising 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. The processes are nonfaulty. The channels are reliable and satisfy the first-in, first-out (FIFO) property. In this paper, we use lowercase English letters (for example, x, y, and z) to denote processes and uppercase English letters (for example, P, Q, and R) to denote sets of processes.

2.2 The Group Mutual Exclusion Problem

The *group mutual exclusion problem (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, which 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 GME problem. However, such algorithms are suboptimal 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 GME should satisfy the following desirable property:

- **Concurrent entry (nontriviality).** 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 eventually be able to enter its critical section. The GME problem can be cast as a *congenial talking philosophers* (CTP) problem [21]. 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 forums. 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.

2.2.1 Complexity Measures

Traditionally, the following metrics have been used to measure the performance of a GME algorithm:

- *message complexity*—the number of messages exchanged per request for a critical section,
- *bit-message complexity*—the number of bits exchanged per request for a critical section,
- *synchronization delay*—the amount of time elapsed between when the current forum terminates and when the next forum (of some other type) can commence,
- *waiting time*—the amount of time elapsed between when a process issues a request for a critical section and when it actually enters the critical section, and
- *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 GME algorithm. We measure synchronization delay and waiting time in terms of the *number of message hops* rather than in terms of real time. Further, as in [12], when analyzing the performance of a GME algorithm theoretically, we compute its *minimum* synchronization delay, *minimum* waiting time, and *maximum* concurrency.

Message complexity and bit-message complexity together capture the overhead imposed on the communication network by the GME algorithm during runtime.

Synchronization delay captures the effectiveness of a GME algorithm in utilizing the resource that it is managing. Intuitively, the minimum synchronization delay measures the amount of time that a *resource has to stay idle* not because of lack of requests for accessing the resource but, rather, because of the algorithm itself. To compute the minimum synchronization delay, it is typically assumed that the system is heavily loaded, and there is a lot of contention among the processes for accessing the resource. Synchronization delay and concurrency together capture the system throughput that can be achieved when the system is heavily loaded (the lower the synchronization delay and the higher the concurrency, the greater the system throughput). A simple way to measure the concurrency of a GME is to compute its maximum concurrency, which is given by the maximum number of processes that can execute their critical sections concurrently.

Waiting time captures the effectiveness of a GME algorithm in fulfilling a request for the resource that it is managing. Intuitively, the minimum waiting time measures the amount of time that a *process with an outstanding request has to wait* not because of other requests for the resource but, rather, because of the algorithm itself. To compute the minimum waiting time, it is typically assumed that the system is lightly loaded, and there is no contention among the processes for accessing the resource.

3 Background

3.1 A 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* $\mathcal{C}$, also referred to as a *coterie*, for a (traditional) mutual exclusion is a set of quorums satisfying the following properties:

- **Intersection:** $\forall P, Q \in \mathcal{C} :: P \cap Q \neq \emptyset$.
- **Minimality:** $\forall P, Q \in \mathcal{C} : P \neq Q : P \not\subseteq Q$.

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 from which x can receive requests for the critical section. We refer to B_x as the *membership set of* x. For a grid quorum system, for every node x, $|B_x| = O(\sqrt{n})$.

Some existing quorum-based algorithms for GME use a special type of quorum system called the *group quorum system* [12]. In a group quorum system, any two quorums belonging to the same group need not intersect, whereas 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. Hereafter, the phrase "quorum system" is used to refer to "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 [3] is used to assign a time stamp to every request for a critical section. A request with a smaller time stamp has a *higher priority* than a request with a larger time stamp (ties are broken using process identifiers). Maekawa's algorithm works as follows:

1. When a process wishes to enter a 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 time stamps to determine whether the process currently holding a lock on it (hereafter referred to as the *locking process*) should be preempted. If 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 member, unlocks the member 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 [22] proved that the message complexity of the above algorithm is $O(q)$, where q is the maximum size of a quorum. Further, its synchronization delay and waiting time are both two message hops. The synchronization delay is two message hops because once a process leaves the critical section, another process can enter the critical section after all the quorum members of the former process have received a RELEASED message and the latter process has received a LOCKED message from its quorum members. (When analyzing the synchronization delay of a quorum-based algorithm derived from Maekawa's algorithm, we ignore the delay incurred due to an exchange of deadlock avoidance messages. This is consistent with the practice of other researchers [22], [12].) Likewise, the waiting time is two message hops because if a process generates a request when there is no other request in the system, then the requesting process can enter the critical section within two message hops after all its quorum members have received its REQUEST message and it has received a LOCKED message from all its quorum members.

4 A Surrogate-Quorum-Based Algorithm

We now describe our approach for solving the GME problem. We call two requests as *compatible* if they are for the same 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 a low message complexity and a low bit-message complexity. To that end, we choose a quorum-based approach. Our algorithm should be efficient, which means that it should have a low waiting time, a low synchronization delay, and a 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 the nodes to issue multiple locks [12]. However, in this approach, deadlock avoidance may require multiple locks to be preempted, thereby increasing the message complexity. To ensure scalability, we take the *leader-follower* approach introduced in [21], 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 requests 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 the GME 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; hence the name *surrogate-quorum*. To avoid repetition, we describe only our extensions to Maekawa's algorithm:

1. A node, when sending a LOCKED message to a process, piggybacks all requests currently in its queue, which 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 that made these requests.
3. A process, on receiving an INVITE message for its current request, sends a CANCEL message to all its quorum members. 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 that it sent.

Since a process 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, INQUIRE, and INVITE messages due to its stale requests. Each of these messages can be piggybacked with the time stamp of the request for which they were generated. 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 it a stale INVITE message. Stale LOCKED, FAILED, and INQUIRE messages should be ignored.

If a leader leaves its forum and 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 on a LOCKED message, the size of a LOCKED message is bounded by $O(br)$, where b is the maximum size of a membership set, and r is the maximum size of a request. Therefore, the worst case bit-message complexity of the modified algorithm is given by $O(bqr)$.

4.2 Reducing Synchronization Delay

A synchronization delay has a significant impact on efficiency, especially the 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 the GME property, we make the following modifications:

1. A leader, upon entering a forum, sets its weight to 1.
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 that it received and the INVITE message, to *all the quorum members* of its *leader*.
5. A node accumulates all the weights that it received over RELEASED messages and maintains its lock until its cumulative weight becomes 1.

An efficient solution for weight distribution and recovery using rational numbers was proposed in [24]. It is clear that a fulfilled request may receive at most q stale INVITE messages, one corresponding to each of its 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 the message complexity to $O(q)$ while maintaining the bit-message complexity at $O(bqr)$. The main idea is to eliminate stale INVITE messages completely.

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 a LOCKED message only after receiving RELEASED messages from all the processes in the current forum. The "new" leader, due to the intersection property of a quorum system, has to obtain a LOCKED message from at least one quorum member of an "old" leader. Hence, there exists a causal path from all the processes leaving a forum to a process entering a forum later as a leader. We exploit this causal path to pass on information about stale requests to the next leader. Specifically, we propose the following changes:

1. Each process maintains a list of stale requests: There is at most one entry in the list for every process in the system. Likewise, each node also maintains a similar list of stale requests (observe that the list maintained by process x is physically different from the list maintained by node x, and the two lists are updated independently). The entry for process y in the list at process (respectively, node) x contains the *latest* request by y that has been fulfilled according to process (respectively, node) x's knowledge.
2. A node, on receiving a RELEASED message from a process, updates its list with the request that sent the RELEASED message.
3. Upon sending a LOCKED message to a process, a node, in addition to piggybacking compatible requests, also piggybacks those stale requests from its list which have not been previously sent to the process.
4. A process, on receiving a LOCKED message from a node, updates its list by using the stale requests received, along with the LOCKED message (note that the list needs to be updated even if the LOCKED message is stale).
5. Upon successfully locking all the quorum members, a leader desists from sending INVITE message to a process y whose request has a time stamp that is less than or equal to the corresponding entry in its vector.

Note that the list of stale requests at a node is updated only on receiving a RELEASED message, whereas the list of stale requests at a process is updated only on receiving a LOCKED message. 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 bit-message complexity of LOCKED messages. Since every process in the system may concurrently enter a given forum, in the worst case, the size of a LOCKED message may be as large as $O(nr)$. This leads to a relatively high bit-message complexity of $O(nqr)$. We will show later that the bit-message complexity, in fact, amortizes to only $O(bqr)$ over all requests.

4.4 Formal Description of Our Algorithm

We refer to our algorithm as Surrogate. A formal description of Surrogate can be found in Figs. 1, 2, 3, and 4. Figs. 1 and 2 describe the actions (P1-P6) of a process as an entity that generates requests for a critical section. Figs. 3 and 4 describe the actions (N1-N5) of a process as a node, that is, as an entity that manages requests for a critical section. The processes are using Lamport's logical clock algorithm [3], which is not shown in the formal description.

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, GME and starvation freedom.

Note that a request for a 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 will 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, with $\sigma = S_0 e_1 S_1 \ldots S_{i-1} e_i S_i \ldots$. For executions with a finite number of such global states, we assume the existence of a hypothetical *nop* event for no operation, which remains enabled in all states following the last global state. A *nop* event does nothing, and so, in our model, for executions with a finite sequence of global states, the final state is repeated infinitely. On executing an *enabled* event e_i in global state S_{i-1}, the system transitions to global state S_i. We assume that a continuously enabled event is eventually executed.

In this paper, we use lowercase Greek letters (for example, α, β, and γ) to denote requests for a 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 a request α, we say that α sent MSG to y.

5.1 Proving Safety (GME)

For proving the safety property, we use the notation described in Fig. 5. Using the notation in the figure, we state some properties about our algorithm that 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:

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

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

$$locked(x, \alpha, i) \wedge locked(x, \beta, i) \Rightarrow \alpha = \beta. \quad (2)$$

We now prove that our algorithm is safe.

Theorem 1 (Safety property). *Surrogate satisfies the GME 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

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

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

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

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 the

```
Actions for process $x$:

Variables:
    $staleListAtProcess$: list of requests that have already been fulfilled, initially $\emptyset$;

    // the following variables have meaningful values only when there is a currently active request
    $Q$: set of quorum members of the currently active request;
    $noOfFailed$: number of FAILED messages that have been sent to the currently active request;
    $inviteSet$: set of requests that are compatible with the currently active request;
    $weight$: weight being held by the currently active request;

Useful Expressions:
    $process(\alpha) \triangleq$ the process to which the request $\alpha$ belongs;

(P1) On generating a request $\alpha$ for critical section:
    // select a quorum for the request
    // depending on the quorum system, every request by process $x$ may use the same quorum
    $Q :=$ select a quorum for the request $\alpha$;
    $noOfFailed := 0$;
    $inviteSet := \emptyset$;
    send REQUEST$\langle\alpha\rangle$ message to all nodes in $Q$;

(P2) On receiving LOCKED$\langle\alpha, compatibleList, staleList\rangle$ message from node $y$:
    // update the list of stale requests and retain only the latest request for a process
    $staleListAtProcess := staleListAtProcess \cup staleList$;
    if ($\alpha$ is not stale) then
        // update the list of requests to which an INVITE message will be sent in case $\alpha$ is satisfied as leader
        $inviteSet := (inviteSet \cup compatibleList) \setminus staleListAtProcess$;
        if (all nodes in $Q$ have been locked) then
            // request satisfied as a leader
            $weight := 1$;
            // invite processes with compatible requests into the forum
            for each request $\beta$ in $inviteSet$ do
                $weight := weight/2$;
                send INVITE$\langle\beta, weight\rangle$ message to $process(\beta)$;
            endfor;
            enter the critical section;
        endif;
    endif;

(P3) On receiving FAILED$\langle\alpha\rangle$ message from node $y$:
    if ($\alpha$ is not stale) then  $noOfFailed := noOfFailed + 1$;  endif;

(P4) On receiving INQUIRE$\langle\alpha\rangle$ message from node $y$:
    if ($\alpha$ is not stale) and ($noOfFailed > 0$) and (not in critical section) then
        // relinquish the lock on node $y$ to avoid a potential deadlock
        send RELINQUISH$\langle\alpha\rangle$ message to node $y$;
    endif;
```

Fig. 1. Formal description of Surrogate for a process (continued in Fig. 2).

same request. From (3) and (4), $type(\alpha) = type(\gamma) = type(\delta) = type(\beta)$. □

5.2 Proving Liveness (Starvation Freedom)

Our algorithm for GME is derived from Maekawa's algorithm for mutual exclusion. Maekawa himself gave a very informal proof for liveness in his paper [22]. We 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 Fig. 6, in addition to the notation used in the safety proof. Before giving the proof, we formalize the concept of the "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 time stamp of α is smaller than the time stamp of β (ties are broken using process

```
Actions for process $x$ (continued):

(P5) On leaving the critical section for request $\alpha$:
    // inform all the quorum members and return the weight
    send RELEASED$\langle \alpha, weight \rangle$ message to all nodes in $Q$;

(P6) On receiving INVITE$\langle \alpha, w \rangle$ message from process $y$:
    // request satisfied as a follower
    $weight := w$;
    send CANCEL$\langle \alpha \rangle$ message to all nodes in $Q$;
    // switch to the quorum of the leader
    $Q :=$ quorum for the request that sent the INVITE message;
```

Fig. 2. Formal description of Surrogate for a process (continued from Fig. 1).

identifiers), and 3) α is waiting for a lock that was sent to β. The case when α has a lower priority than β can be easily resolved to one of the cases above. Formally,

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

Based on the above definition of the 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 $WFG(i)$. The following properties about a wait-for graph can be easily verified.

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 $|\Pi|$.*

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 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,

$$\begin{aligned} &pending(\alpha, i) \wedge \langle \forall x : x \in quorum(\alpha) : \alpha \in queue(x, i) \rangle \\ &\Rightarrow \\ &\langle \forall u, v : u, v \geq i : u \leq v \Rightarrow wait\text{-}set(\alpha, u) \supseteq wait\text{-}set(\alpha, v) \rangle. \end{aligned} \tag{5}$$

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 a higher priority (that is, smaller time stamp) 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 the pending requests in S_i.

Note that a request may be involved in two kinds of wait relationships: 1) It may be waiting on a request with a smaller time stamp than its own, or 2) it may be waiting on a request with a larger time stamp than its own. Intuitively, the former kind of wait is *desirable*, whereas the latter kind of wait is *undesirable*. The notions of the "waiting-on relation" and "wait-for graph" capture the undesirable wait relationships. On the other hand, the notions of "potence" and "omnipotence" capture the desirable wait relationships.

Our liveness proof consists of the following steps: First, we show that the wait-set of a pending request eventually becomes permanently empty (that is, eventually, all undesirable wait relationships of a pending request disappear permanently). Second, we show that a pending request eventually becomes potent and then omnipotent (that is, eventually, all desirable wait relationships of a pending request also disappear permanently). Finally, we show that once a request becomes omnipotent, and its wait-set becomes permanently empty, it is eventually satisfied.

5.2.1 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, then it simply ignores the message. If that happens, then the 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 consisting of RELEASED followed by LOCKED messages from all the 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 the 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,*

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

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 that it received

```
Actions for node $x$:

Variables:
    $queue$: queue of outstanding requests, initially $\emptyset$;
    $staleListAtNode$: list of requests that have already been fulfilled, initially $\emptyset$;
    $lockedBy$: request that currently holds the lock, initially $\perp$;
    $weight$: weight that has been recovered so far, initially 0;

(N1) On receiving REQUEST$\langle\alpha\rangle$ message from process $y$:
    if ($lockedBy = \perp$) then
        add $\alpha$ to $queue$;
        call sendLockedMessage( );
    else
        // the lock has been issued to another request
        let $\beta$ be the request with the smallest timestamp in $queue$;
        add $\alpha$ to $queue$;
        if ($\beta$'s timestamp is smaller than $\alpha$'s timestamp) then
            // some other pending request has higher priority than $\alpha$
            send FAILED$\langle\alpha\rangle$ message to $y$;
        else
            if ($\beta = lockedBy$) then
                // ask $process(\beta)$ to relinquish the lock
                send INQUIRE$\langle\beta\rangle$ message to $process(\beta)$, if not already sent;
            else
                // inform $process(\beta)$ that another request has higher priority than its request
                send FAILED$\langle\beta\rangle$ message to $process(\beta)$, if not already sent;
            endif;
        endif;
    endif;

(N2) On receiving CANCEL$\langle\alpha\rangle$ message from process $y$:
    // the request has been satisfied as a follower
    if ($lockedBy = \alpha$) then   $lockedBy := \perp$;   endif;
    remove $\alpha$ from $queue$;
    call sendLockedMessage( );

(N3) On receiving RELEASED$\langle\alpha, w\rangle$ message from process $y$:
    // update the list of stale requests and retain only the latest request for a process
    $staleListAtNode := staleListAtNode \cup \{\alpha\}$;
    remove $\alpha$ from $queue$, if present;
    $weight := weight + w$;
    if ($weight = 1$) then
        // all the weight has been recovered
        $lockedBy := \perp$;
        call sendLockedMessage( );
    endif;
```

Fig. 3. Formal description of Surrogate for a node (continued in Fig. 4).

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 $x \in quorum(\alpha) \cap quorum(\beta)$ 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 β if it has not done so already.

```
Actions for node $x$ (continued):

(N4) On receiving RELINQUISH$\langle\alpha\rangle$ message from process $y$:
    // the lock has been released
    $lockedBy := \bot$;
    call sendLockedMessage( );

(N5) On invocation of sendLockedMessage( ):
    if ($lockedBy = \bot$) and ($queue \neq \emptyset$) then
        // the lock is available and there is a pending request in the queue
        let $\alpha$ be the request with the smallest timestamp in $queue$;
        $weight := 0$;
        $lockedBy := \alpha$;
        $compatibleList :=$ set of all requests in $queue$ that are compatible with $\alpha$;
        $staleList :=$ set of requests in $staleListAtNode$ that have not yet been sent to the process
                  to which $\alpha$ belongs;
        send LOCKED$\langle\alpha, compatibleList, staleList\rangle$ message to $\alpha$'s process;
    endif;
```

Fig. 4. Formal description of Surrogate for a node (continued from Fig. 3).

$\mathcal{R}_i$	$\triangleq$	set of all requests for critical section made in some global state up to (and including) S_i
$process(\alpha)$	$\triangleq$	process to which request α belongs
$type(\alpha)$	$\triangleq$	type of request α
$quorum(\alpha)$	$\triangleq$	quorum selected by request α
$forum(\alpha, i)$	$\triangleq$	$process(\alpha)$ is in a forum at global state S_i and α is the latest request of $process(\alpha)$
$leader(\alpha)$	$\triangleq$	request α is satisfi ed as a leader (assuming α is eventually satisfied)
$follower(\alpha)$	$\triangleq$	request α is satisfi ed 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. 5. Notation used in the safety proof.

$ts(\alpha)$	$\triangleq$	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. 6. Additional notation used in the liveness proof.

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. If the first condition holds before the other, β sends a RELINQUISH message to node x. Otherwise, if 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 $level(\alpha, i)$, 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 $rank(\alpha, i)$, as the maximum value attained by its level in any global state including and following S_i. Formally,

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

Note that the rank of a request is monotonically nonincreasing 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,*

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

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 in 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,

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

The lemma can now be proved using mathematical induction on the rank of a request. □

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

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

5.2.2 An Omnipotent Request Is Eventually Satisfied

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

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

Proof. From (7), the 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. 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. □

5.2.3 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,*

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

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 a 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 time stamp 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 $compete\text{-}set(\alpha, i)$, as the set of all pending requests in global state S_i that have a 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,

$$\langle \forall u, v : i \leq u \leq v : compete\text{-}set(\alpha, u) \supseteq compete\text{-}set(\alpha, v)\rangle. \tag{8}$$

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

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

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_u, then every request in $compete\text{-}set(\alpha, u)$ is also potent in S_u. Consider the request β with the smallest time stamp in $compete\text{-}set(\alpha, u)$. Clearly, β is omnipotent in S_u. Using Lemma 8, β is eventually satisfied and, therefore, eventually leaves the compete set of α. □

5.2.4 Combining the Three

Theorem 10 follows from Lemma 7, Lemma 8, and Lemma 9.

Theorem 10 (Liveness property). *Every request is eventually fulfilled. Formally,*

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

6 Complexity Analysis

In this section, we analyze the performance of our algorithm with respect to the following metrics: message complexity, bit-message complexity 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. Clearly, the number of REQUEST, FAILED, and CANCEL messages are each bounded by q. We call a LOCKED message from a quorum member as *successful* if the locking request does not send any RELINQUISH message to the member after receiving that LOCKED message. Clearly, the number of successful LOCKED messages is bounded by q. An INQUIRE message is only generated for a new request. Hence, at most one INQUIRE message can be generated per request at a given node. Therefore, the number of INQUIRE, RELINQUISH, and unsuccessful LOCKED messages are each bounded by q per request. All that remains to be bounded are the numbers of INVITE and RELEASED messages. Since a follower, upon leaving its forum, sends RELEASED messages to all the 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 next analyze the worst case bit-message complexity of Surrogate. Note that all messages carry one or more requests. Some messages such as INVITE and RELEASED also carry a weight. A leader sends at most one INVITE message to any process. Therefore, using the weight distribution and recovery scheme described in [24], the amount of space required to store a weight is given by $O(\log n)$, which is $O(r)$. Therefore, all messages except LOCKED messages have a size of $O(r)$. The worst-case size of LOCKED messages is $O(nr)$. This implies that the worst case bit-message complexity of our algorithm is $O(nqr)$. We show that the bit-message complexity of our algorithm is only $O(bqr)$ when amortized over all requests. This, in turn, implies that only a small number of LOCKED messages are large and have a size of $\Theta(nr)$.

Theorem 12. *The bit-message complexity of* Surrogate *is $O(bqr)$ when amortized over all requests.*

Proof. All messages except LOCKED messages have a size of $O(r)$. A LOCKED message carries two separate kinds of requests: compatible requests and stale requests. Since a node can only receive requests from processes in 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.

A request can be inserted into a list of stale requests of at most q nodes. This is because a node inserts a request into its list of stale requests only on receiving a RELEASED message from that request, and a request sends at most q RELEASED messages. Now, a node sends a LOCKED message to at most b processes. Therefore, a node piggybacks a request as a stale request on at most b LOCKED messages . This implies that each request generates at most $O(bqr)$ bits piggybacked as a stale request on LOCKED messages.

Let t denote the total number of requests generated in the system. Clearly, the total number of bits exchanged over all messages, when summed over all requests, is given by $O(bqrt)$. Hence, the amortized bit-message complexity of Surrogate is given by $O(bqr)$ per request. □

Note that a follower has to send a RELEASED message to the nodes in its leader's quorum. Depending on the quorum system used, a leader may need to piggyback its entire quorum on any INVITE message that it sends to its followers. However, this does not increase the bit-message complexity of Surrogate because there is at most one INVITE message per request. To analyze the synchronization delay of Surrogate, observe that once a session terminates, the next session is initiated as soon as RELEASED messages for the current session have been received by the quorum members and the leader for the next session has received LOCKED messages from all its quorum members. Therefore, the synchronization delay of Surrogate is given by two message hops corresponding to a RELEASED message followed by a LOCKED message. As far as the waiting time of Surrogate is concerned, if a process generates a request when there is no other request in the system, then it can enter the requested forum within two message hops corresponding to a REQUEST message followed by a LOCKED message (note that when the system is lightly loaded, all requests are satisfied as leaders). Finally, if all processes make compatible requests, then all of them can be in the forum at the same time.

Theorem 13. Surrogate *has a minimum synchronization delay of two message hops, a minimum waiting time of two message hops, and a maximum concurrency of n.*

7 Experimental Evaluation

We experimentally compare the performance of Surrogate with Joung's first algorithm Maekawa_M by using discrete-event simulation. We compare the performance of the two

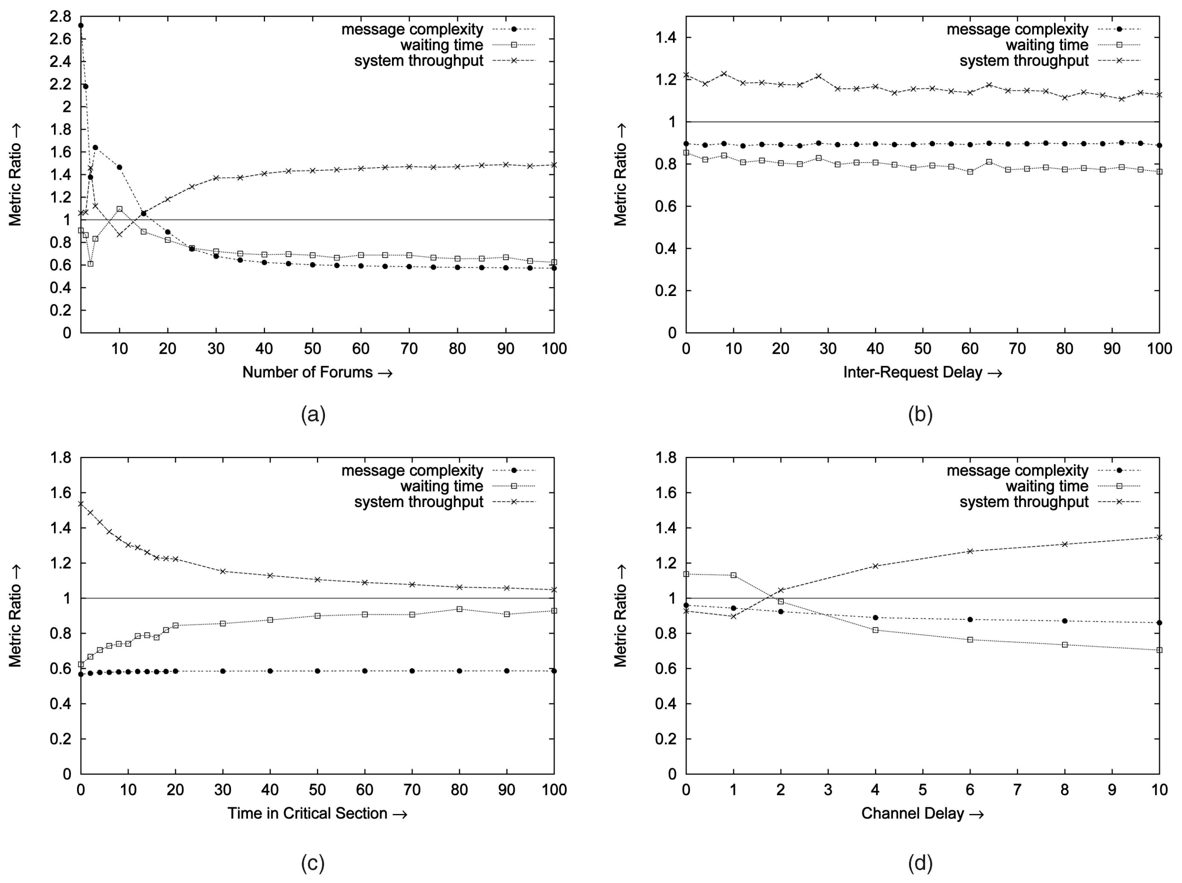

Fig. 7. Relative performance of the two algorithms as a function of various parameters. (a) Varying the number of forums (m) with $\mu_{ncs} = 4$ time units, $\mu_{cs} = 2$ time units, and $\mu_{cd} = 4$ time units. (b) Varying the mean inter-request delay (μ_{ncs}) with $m = 20$, $\mu_{cs} = 2$ time units, and $\mu_{cd} = 4$ time units. (c) Varying the mean time in the 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.

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 $\left(\frac{\text{Surrogate's performance}}{\text{Maekawa\_M's performance}}\right)$ for each metric. Note that for the message complexity and waiting time, a ratio of less than 1 would imply that Surrogate has better performance than Maekawa_M. On the other hand, for the system throughput, a ratio of greater than 1 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 interrequest delay (that is, the duration of the noncritical 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 $[0, 2 * \mu_{cs}]$ (the duration of the critical section). The 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 the number of processes n, which is set to 25, and the number of requests per process, which is set to 1,000. All other parameters are varied one by one to study their effect on the relative performance of the two algorithms.

For Surrogate, we use the grid quorum system [22]. For Maekawa_M, the maximum number of simultaneous locks that a node can grant is set to the maximum, which is n. This maximizes the average concurrency of Maekawa_M, which, in turn, lowers the average waiting time of a request. To construct the surficial quorum system, $\sqrt{\frac{2n}{m(m-1)}}$ has to be an integer. If not, then 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 manner.

7.1 Simulation Results

Fig. 7 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 a 95 percent confidence level.

As our simulation results demonstrate, when the number of forums is small, or the mean channel delay is very low, Maekawa_M has better performance than Surrogate. In almost all other cases, Surrogate has better performance than Maekawa_M. Specifically, the message complexity and waiting time decrease by as much as 40 percent, and the system throughput increases by as much as 48 percent.

Further, the performance gap between the two algorithms increases (in favor of Surrogate) as either 1) the number of forums increases (Fig. 7a) or 2) the ratio of the mean channel delay to the mean time in the critical section increases (Figs. 7c and 7d).

The reasons for the relative behavior of the two algorithms are given as follows: When the number of forums is small, the surficial quorum system has smaller quorums than the grid quorum system and, moreover, has a high degree. For example, when $m = 2$, each quorum in the surficial quorum system is of size $\sqrt{n}$, whereas each quorum in the grid quorum system is of size $2\sqrt{n} - 1$. Also, when $m = 2$, the degree of the surficial quorum system is $\sqrt{n}$. Not surprisingly, Maekawa_M has better performance than Surrogate for a small number of forums.

As the number of forums increases, the degree of the surficial quorum system decreases. Further, the fraction of requests with which a request conflicts also increases. For example, when $m = 2$, a request conflicts with only 50 percent of the requests on the average. On the other hand, when $m = 10$, a request conflicts with as many as 90 percent of the requests on the average. In Surrogate, a process with an outstanding request enters the forum as a leader once it has successfully locked all its quorum members. It then invites other processes with compatible requests to enter the forum. Intuitively, only requests satisfied as leaders *compete* with other requests to lock their respective quorum members. Requests satisfied as followers stop competing with other requests once they receive an INVITE message. In Maekawa_M, on the other hand, every request is satisfied as a "leader" in the sense that a process with an outstanding request can enter the forum only after it has successfully locked all its quorum members. Clearly, there is a higher contention among conflicting requests to lock their respective quorum members in Maekawa_M as compared to Surrogate. As a result, Maekawa_M has a higher probability of "priority inversion," in which a request with a smaller time stamp is forced to wait on a request with a larger time stamp. Whenever priority inversion occurs, Maekawa_M exchanges deadlock avoidance messages (namely, INQUIRE and RELINQUISH), which may force a large number of processes to relinquish their locks. This translates into a higher average delay between the termination of a session and the commencement of the next session in Maekawa_M due to this exchange of deadlock avoidance messages. Furthermore, this delay increases as the ratio of the mean channel delay to the mean time in the critical section increases. Another reason for the better performance of Surrogate may be the addition of new (logical) processes so that the surficial quorum system can actually be constructed. However, even when Maekawa_M is used with the grid quorum system (Toyomura et al.'s algorithm), our simulation results show that Surrogate still continues to have much better performance, especially when the mean channel delay is large.

8 Discussion

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

8.1 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 the requests are of the same type. However, we can achieve a 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 member to the locking process continues until the node learns about a pending conflicting request. Specifically, a 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 conflicting 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 member may continue forwarding 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.

Note that, once a quorum node has recovered its lock, it can safely assume that any request that is still in its queue of outstanding requests was not sent an INVITE message and, therefore, is still pending, even if that request was piggybacked on a FORWARD message. If the request was indeed sent an INVITE message, then the quorum node can recover its lock only after receiving a RELEASED message from that request, which, in turn, causes the request to be removed from the queue. Clearly, with the above modifications, our algorithm satisfies the concurrent entry property. If all the 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, the leader of the current forum receives a STEPDOWN message within two message hops.

With regard to the message complexity, note that a FORWARD or a STEPDOWN message is generated only for a new request arriving at a 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 that it receives, implying that the message complexity is still $O(q)$. The synchronization delay remains at two message hops because when the system is heavily loaded, the STEPDOWN message is piggybacked on the LOCKED message itself. The waiting time also does not increase. When the system is lightly loaded, all the requests are satisfied as leaders. Specifically, a process entering the forum as a leader does not have any followers and releases its locks on quorum members soon after leaving the forum. However, with the above-described modification, a leader may invite a process into its forum more than once, which implies that the amount of space required to store a weight is no longer $O(r)$. Therefore, the bit-message complexity is now given by $O((br + w)q)$, where w denotes the maximum amount of space required to store a weight. Our experimental results show that the modification for the concurrent entry does not result in a significant improvement in the performance.

8.2 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 the data has been replicated. In the traditional GME, a process has to specify the type of the critical section that it wants to execute at the time of the request, which translates into specifying the CD that it wants to access for satisfying its request. This may lead to an *unnecessary delay (or blocking)* in satisfying a request. To eliminate this unnecessary delay, Manabe and Park extend the GME problem in [23] 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 the 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 $type\text{-}set(\alpha)$ 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 $type\text{-}set(\alpha) \cap type\text{-}set(\beta) \neq \emptyset$. 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 $intersecting\text{-}set(\alpha)$ denote the set of requests that it has received from its quorum members, along with LOCKED messages. Note that $intersecting\text{-}set(\alpha)$ may contain two requests β and γ such that 1) $type\text{-}set(\alpha) \cap type\text{-}set(\beta) \neq \emptyset$, 2) $type\text{-}set(\alpha) \cap type\text{-}set(\gamma) \neq \emptyset$, and 3) $type\text{-}set(\beta) \cap type\text{-}set(\gamma) = \emptyset$. Clearly, α cannot send INVITE messages to both β and γ. To maximize the concurrency, α chooses a type $t \in type\text{-}set(\alpha)$ for which the number of requests in $intersecting\text{-}set(\alpha)$ that are "compatible" with t is *maximized*. All the complexity measures remain the same except for the bit-message complexity, which increases by a factor of s, where s is the maximum number of types that a process can specify at the time of making a request.

9 Conclusion

We have proposed an efficient distributed algorithm Surrogate for solving the GME problem based on the notion of *surrogate-quorum*. Unlike the existing quorum-based algorithms for GME [12], 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. This is achieved at the expense of the bit-message complexity, which increases to $O(bqr)$ per request. Further, unlike the algorithms in [12], which assume that the number of groups 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, which is another quorum-based GME algorithm, in almost all cases except when the number of forums is small, and the mean channel delay is low. We have also presented two extensions to our basic algorithm for achieving other desirable properties.

A natural extension of GME 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 one 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.

Appendix

Omitted Proofs

Proof of Lemma 2. The first property follows from the fact that the time stamp value along any path increases monotonically. The second property follows from the first property. The third property follows from the definition of the 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. □

Proof of Lemma 6. We prove the lemma by using mathematical induction on the rank of a request:

- **Base case** $(\mathbf{rank}(\alpha, \mathbf{i}) = \mathbf{0})$. Clearly, $wait\text{-}set(\alpha, u)$ is empty for all $u \geq i$. Using (6), the property holds.
- **Induction step** $(\mathbf{rank}(\alpha, \mathbf{i}) = \mathbf{k} \textbf{ with } \mathbf{k} > \mathbf{0})$. From (5), eventually, the system reaches a global state S_u, after which the wait-set of α stops growing. Consider a request β in $wait\text{-}set(\alpha, u)$. 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 $rank(\beta, u) < rank(\alpha, u) \leq rank(\alpha, i)$. 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 that the wait-set of α eventually becomes empty. Therefore, using (6), the property holds.

This establishes the lemma. □

ACKNOWLEDGMENTS

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

REFERENCES

[1] R. Atreya and N. Mittal, "A Dynamic Group Mutual Exclusion Algorithm Using Surrogate-Quorums," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS '05)*, pp. 251-260, June 2005.

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

[3] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, 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," *Comm. ACM*, vol. 24, no. 1, pp. 9-17, Jan. 1981.

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

[6] K. Raymond, "A Tree Based Algorithm for Distributed Mutual Exclusion," *ACM Trans. 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)," *Proc. 20th Ann. Symp. Foundations of Computer Science (FOCS '79)*, pp. 234-254, Oct. 1979.

[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 Trans. Programming Languages and Systems*, 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*, vol. 13, no. 4, pp. 189-206, 2000.

[12] Y.-J. Joung, "Quorum-Based Algorithms for Group Mutual Exclusion," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 5, pp. 463-475, May 2003.

[13] M. Toyomura, S. Kamei, and H. Kakugawa, "A Quorum-Based Distributed Algorithm for Group Mutual Exclusion," *Proc. Fourth Int'l Conf. Parallel and Distributed Computing, Applications and Technologies (PDCAT '03)*, pp. 742-746, Aug. 2003.

[14] P. Keane and M. Moir, "A Simple Local-Spin Group Mutual Exclusion Algorithm," *Proc. 18th ACM Symp. Principles of Distributed Computing (PODC '99)*, pp. 23-32, 1999.

[15] V. Hadzilacos, "A Note on Group Mutual Exclusion," *Proc. 20th ACM Symp. Principles of Distributed Computing (PODC '01)*, Aug. 2001.

[16] K. Vidyasankar, "Brief Announcement: A Highly Concurrent Group Mutual L-Exclusion Algorithm," *Proc. 21st ACM Symp. Principles of Distributed Computing (PODC '02)*, p. 130, July 2002.

[17] K. Vidyasankar, "A Simple Group Mutual L-Exclusion Algorithm," *Information Processing Letters*, vol. 85, no. 2, pp. 79-85, 2003.

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

[19] S. Cantarell, A.K. Datta, F. Petit, and V. Villain, "Group Mutual Exclusion in Token Rings," *Computer J.*, vol. 48, no. 2, pp. 239-252, 2005.

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

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

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

[23] Y. Manabe and J. Park, "A Quorum-Based Extended Group Mutual Exclusion Algorithm without Unnecessary Blocking," *Proc. 10th Int'l Conf. Parallel and Distributed Systems (ICPADS '04)*, pp. 341-348, 2004.

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

Ranganath Atreya received the bachelor's degree, with distinction, from Bangalore University, India, in 2000 and the MS degree in computer science from the University of Texas at Dallas in 2004. He has been working at Amazon.com since January 2005. He was the recipient of the National Award for the Best Bachelor of Engineering Project in 2000.

Neeraj Mittal received the BTech degree in computer science and engineering from the Indian Institute of Technology, Delhi, in 1995 and the MS and PhD degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor at the Department of Computer Science and a codirector of the Advanced Networking and Dependable Systems Laboratory (ANDES), University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking, and databases. He is a member of the IEEE Computer Society.

Sathya Peri received the MCSA 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 the PhD degree in computer science at Advanced Networking and Dependable Systems Laboratory (ANDES), University of Texas at Dallas. His research interests include peer-to-peer computing and dynamic distributed systems.