AN EFFICIENT DISTRIBUTED GROUP MUTUAL EXCLUSION ALGORITHM FOR NON-UNIFORM GROUP ACCESS

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

Prajwal K. Mohan*
Digital Home Group
Intel Corporation
Hillsboro, OR 97124, USA
email: prajwal.karur.mohan@intel.com

ABSTRACT

In the group mutual exclusion problem, each critical section has a *type* or a *group* associated with it. Processes requesting critical sections of the same type may execute their critical sections concurrently. However, processes requesting critical sections of different types must execute their critical sections in a mutually exclusive manner.

Most algorithms for group mutual exclusion that have been proposed so far implicitly assume that all groups are equally likely to be requested. In this paper, we propose an efficient algorithm for solving the problem when a relatively small number of groups are requested more frequently than others. Our algorithm has a message complexity of $2n - 1$ per request for critical section, where n is the number of processes in the system. It has low synchronization delay of t and low waiting time of $2t$, where t denotes the maximum message delay. The maximum concurrency of our algorithm is n, which implies that if all processes have requested critical sections of the same type, then all of them may execute their critical sections concurrently. Finally, the *amortized* message overhead of our algorithm is $O(1)$.

Our experimental results indicate that our algorithm *outperforms* the existing algorithms by as much as 50% in some cases.

KEY WORDS

message-passing system, resource management, group mutual exclusion, token-based algorithm, non-uniform group access

1 Introduction

Resource management is one of the most important and fundamental problems in distributed systems. Typically, to maintain the integrity of a resource, at most one process should access the resource at any time. As a result, accesses to the same resource (that is, execution of critical sections) by different processes have to be serialized. This problem is referred to as the *mutual exclusion problem*. Mutual exclusion has been studied extensively and a large number of solutions have been developed (*e.g.*, [1, 2, 3, 4, 5]). Many different variants of mutual exclusion have also been defined. Some examples of variants include k-mutual exclusion problem [6], dining philosophers problem [7] and drinking philosophers problem [8].

Recently, Joung proposed another variant of the mutual exclusion problem called the *group mutual exclusion problem* [9]. In the group mutual exclusion (GME) problem, every critical section is associated with a *type* or a *group*. Critical sections belonging to the same group can be executed concurrently. However critical sections belonging to different groups must be executed in a mutually exclusive manner. Intuitively, if two critical sections belong to the same group, then their requesting processes share some common property. The multiple-readers/single-writer (MRSW) problem is a special case of the group mutual exclusion problem. Let n denote the number of processes in the system. The multiple-readers/single-writer problem can be modeled using $n + 1$ groups. All read critical sections belong to the same group. On the other hand, write critical sections requested by different processes belong to different groups. (There is one group for each process.) As another application of the problem, assume that data is stored on CDs in a CD-jukebox and only one disk can be loaded for access at any time [9]. Clearly, when a disk is loaded, users that need data on the currently loaded disk can access the disk concurrently. On the other hand, users that need access to a different disk have to wait until no one is using the currently loaded disk. Manabe and Park [10] proposed an extension of group mutual exclusion in which, at the time of request, a process is allowed to specify more than one type; the request can be fulfilled as long as the process gets to execute a critical section of any one of those types. This corresponds to the case when the same data is replicated on multiple CDs and any of those CDs can be used to satisfy a request for that data.

Most algorithms that have been developed for group mutual exclusion so far [9, 11, 12, 13, 14, 15] implicitly assume that all groups are *equally likely* to be requested whenever a request for critical section is generated. In a real world, however, probability of requesting a group may be non-uniformly distributed. Specifically, a small fraction of the groups may be requested more frequently than the rest of the groups. For instance, in a CD jukebox, few CDs may contain data that are accessed more frequently than data on other CDs. As another example, in the MRSW problem, read requests may be more common than write

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

requests. In this paper, our objective is to develop an efficient distributed algorithm for solving the group mutual exclusion problem that exploits the fact that group access requests *may not be* uniformly distributed.

Related Work: In [16], Joung modified Ricart and Agrawala's algorithm for traditional mutual exclusion [2] to derive two algorithms for group mutual exclusion (RA1 and RA2). The message-complexity of the two algorithms (RA1 and RA2) is $2(n-1)$ messages and $3(n-1)$ messages (amortized over all requests), respectively, where n is the number of processes in the system. Their synchronization delay, typically measured when the system is heavily loaded, is at most t, where t is the maximum message delay. And, their waiting time, typically measured when the system is lightly loaded, is at most $2t$. Moreover, both algorithms have maximum concurrency of n, which implies that if all processes have requested critical sections of the same type, then it is possible for all of them to execute their critical sections concurrently. However, RA1 has low expected concurrency of $O(1)$ under heavy loads, whereas RA2 has high message overhead of $O(n)$. (Each message in RA2 carries a vector timestamp of size n.)

In [14], Joung proposed two quorum-based algorithms for group mutual exclusion (Maekawa_M and Maekawa_S), which are based on Maekawa's quorum-based algorithm for traditional mutual exclusion. He also proposed a quorum system suitable for group mutual exclusion called the *surficial quorum system*. His first algorithm Maekawa_M has high message complexity of $O(n \cdot m)$ messages per request for critical section in the worst-case, where m is the number of different groups, but preserves the synchronization delay of the original Maekawa's algorithm. His second algorithm Maekawa_S has message complexity of $2q+1$, where q is the maximum size of a quorum. However, its synchronization delay is $(q+1)t$, which is quite high. In [15], Atreya and Mittal proposed another quorum-based group mutual exclusion algorithm using the notion of surrogate quorums.

Algorithms for group mutual exclusion problem have also been developed for ring networks (processes are arranged in a ring) [12, 11] and tree networks (processes are arranged in a tree) [13]. However, all these solutions have high synchronization delay and high waiting time.

Our Contribution: In this paper, we propose a highly concurrent token-based algorithm for group mutual exclusion, especially suited for applications in which a relatively small fraction of groups is requested much more frequently than the rest of the groups. Our algorithm is derived from Suzuki and Kasami's token-based algorithm for traditional mutual exclusion [4]. We achieve concurrency by using two kinds of tokens, namely *primary* and *secondary*. At any time, there is only one primary token but there may be multiple secondary tokens. Our algorithm has a message-complexity of $2n-1$ messages per request for critical section. It has low synchronization delay of at most t and low waiting time of at most $2t$. The maximum concurrency of our algorithm is n. Further, our algorithm has high expected concurrency under heavy loads (as in the case of RA2) and low *amortized* message overhead of $O(1)$ (as in the case of RA1). Our experimental results indicate that our algorithm *significantly outperforms* the existing group mutual exclusion algorithms with respect to all the three important complexity measures, namely message-complexity, waiting time and system throughput, by as much as 50% in some cases.

Paper Organization: This paper is organized as follows. We present our system model and formally describe the group mutual exclusion problem in Section 2. Our token-based algorithm for group mutual exclusion is discussed in Section 3. We present our experimental results in Section 4. Finally, we present our conclusions and outline future research directions in Section 5.

2 Model and Problem Definition

2.1 System Model

We assume an asynchronous message-passing distributed system comprising of a set of n processes $P = \{p_1, p_2, \ldots, p_n\}$, 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 and channels are reliable. 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 by Joung in [9] as an extension of the traditional mutual exclusion problem. In this problem, every request for a critical section is associated with a *type* or a *group*. Any algorithm for group mutual exclusion should satisfy the following properties:

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

Intuitively, if two critical sections belong to the same group, then their requesting processes share some common property. The multiple-readers/single-writer problem is a special case of the group mutual exclusion problem. Clearly, any algorithm for solving the traditional mutual exclusion problem also solves the group mutual exclusion problem. However, such algorithms are sub-optimal because they force all critical sections to be executed in a mutually exclusive manner and therefore do not permit any concurrency whatsoever. To avoid such degenerate solutions and unnecessary synchronization, Joung proposed

that an algorithm for achieving group mutual exclusion should satisfy the following desirable property:

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

We use m to denote the number of different groups. For the multiple-readers/single-write problem, $m = n + 1$.

For convenience, we use the term *session* to refer to a time-interval in which all critical sections executed are of the same type. A session commences with the start of the first critical section and terminates with the end of the last critical section.

2.2.1 Complexity Measures

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

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

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

3 A Token-Based Distributed Algorithm for Group Mutual Exclusion

In this section, we describe our token-based algorithm for group mutual exclusion. Our algorithm is an extension of the Suzuki and Kasami's algorithm for traditional mutual exclusion [4]. (Note that any algorithm that solves the group mutual exclusion problem also solves the mutual exclusion problem. Therefore, not surprisingly, almost all algorithms for group mutual exclusion have been obtained by modifying some mutual exclusion algorithm.) For the sake of completeness, we provide a brief description of the Suzuki and Kasami's algorithm as well. We also prove the correctness of our algorithm and analyze its performance. Due to lack of space, a formal description of our algorithm and proofs of various lemmas and theorems are given elsewhere [17].

3.1 The Background: Suzuki and Kasami's Token-Based Algorithm

Suzuki and Kasami's algorithm [4] achieves mutual exclusion by maintaining a *unique* token in the system. A process can enter the critical section only if it holds the token. The token is said to be busy if its holder is currently in the critical section; otherwise it is *idle*. Every process maintains a vector containing the sequence number of the latest request made by each process that it is aware of. The token contains a first-in-first-out (FIFO) queue of process that have requests waiting to be fulfilled. Additionally, the token also contains a vector that maintains the number of requests that have been fulfilled for each process. A process, on generating a request for critical section, sends a REQUEST message to other processes if it does not have the token already. The process holding the token, on learning about a pending request, sends the token to the requesting process (via a TOKEN message) as soon as the token becomes idle. The message-complexity of Suzuki and Kasami's algorithm is n messages per request for critical section. Its synchronization delay is at most t and its waiting time is at most $2t$. The amortized message overhead is $O(1)$ because, for each request for critical section, there are $n - 1$ REQUEST messages of size $O(1)$ and one TOKEN message of size $O(n)$.

3.2 Our Algorithm

To enhance concurrency of Suzuki and Kasami's basic algorithm, we use multiple tokens. There are two kinds of tokens: *primary* and *secondary*. At any time, there is exactly one primary token in the system. However, the number of secondary tokens may vary from time-to-time. Initially, process p_1 has the primary token, and the number of secondary tokens is zero.

A token has a type associated with it and it can only be used to enter critical section of that type. Similar to Suzuki and Kasami's algorithm, a process can enter a critical section of certain type only if it holds a token—primary or

secondary—of that type. The difference between a primary and a secondary token is the following: A process holding a primary token is allowed to issue secondary tokens to other processes. However, a process holding a secondary token is not allowed to issue a token to another process.

A process, on requesting a critical section, checks whether it has a token of the same type as the critical section requested. If yes, it can simply enter the critical section. If not, it sends a REQUEST message to all processes.

Every process p_i maintains a vector $request_i$; the jth entry of $request_i$ contains the sequence number of the latest request of process p_j, along with its type, that process p_i knows of. A token—primary as well as secondary—contains a vector $fulfilled$; the jth entry of $fulfilled$ captures the number of requests of process p_j that have been fulfilled so far as per the token. If a process p_i holds a token, then it can determine whether a process p_j has a pending request by comparing the jth entries of $request_i$ and $token_i.fulfilled$: p_j has a pending (or new) request if $request_i[j].number > token_i.fulfilled[j]$. The process holding the primary token, on receiving a "new" request for critical section of the same type as the type of the token, issues a secondary token to the requesting process. Additionally, a primary token also contains a queue of pending requests. In case a request cannot be fulfilled immediately, which happens if its type conflicts with that of the token, the request is stored in the token queue.

A process holding a token keeps the token until it becomes aware of a *conflicting* pending request, that is, pending request for critical section of type that is different from the type of the token. Specifically, a process holding a secondary token, on learning about a conflicting pending request, releases the token by sending a RELEASE message to all processes once it is no longer in the critical section (that is, the token is idle). On the other hand, the process holding the primary token, on learning about a conflicting pending request, passes the primary token to another process, which has a pending request in the token queue, once the token becomes idle. To select the next primary token holder, we first determine the *type* (or group) of the next session that should be initiated using a priority-based scheme discussed later in Section 3.2.1. Once the type has been determined, one of the processes that has requested a critical section of the selected type is chosen to become the next primary token holder. All other processes with pending requests for the selected type are issued secondary tokens.

A process p_j on receiving a token from the previous primary token holder p_i may not be able to use the token immediately. In other words, the token may not be *safe* for use. This is because a non-zero number of secondary tokens may have been issued in the previous session and some of these tokens may still be busy. Clearly, process p_j should wait until these tokens have been released before it can use its token to enter the critical section. To that end, we associate a sequence number with each token that represents the session to which the token belongs (given by variable $session$ in the token). A process, on releasing a secondary token, piggybacks this sequence number on the RELEASE message it sends. Each process records the number of RELEASE messages it has received containing the most recent session sequence number (given by variables $numRelease_j$ and $session_j$ for process p_j). Further, a token contains the number of secondary tokens that were issued for the previous session (given by variable $oldNumTokens$ in the token). To determine if a token it has received is safe for use, process p_j evaluates the following condition:

$$\begin{aligned}&(token_j.oldNumTokens = 0) \bigvee \\ &((session_j = token_j.session - 1) \wedge \\ &\ (numRelease_j = token_j.oldNumTokens))\end{aligned} \tag{1}$$

Clearly, if no secondary tokens were issued in the previous session, then the token is safe for use. Otherwise, the token is safe for use if process p_j has received release messages for all secondary tokens issued in the previous session. Additionally, the token is also safe for use if process p_j has received a RELEASE message for the current session from some other process, say p_k. This implies that process p_k deemed it safe to enter the critical section. This could have happened only if process p_k received all RELEASE messages for the previous session or was told by some other process (via a RELEASE message) that it was safe to use the token. Therefore we modify the condition for testing for the safety of a token as follows:

$$(session_j = token_j.session) \vee (1) \tag{2}$$

Note that, in our algorithm, the relationship between a primary token holder and a secondary token holder of the same session is *different* from that of a captain and its follower in RA2 [16]. Specifically, in our algorithm, it is possible for a secondary token holder to enter the critical section and perhaps even leave it before the primary token holder is able to enter the critical section. This may happen, for example, when a process receives a secondary token from the primary token holder of the previous session before the token arrives at the primary token holder of the current session.

3.2.1 Criterion for Selecting the Type of the Next Session

To minimize the waiting time, it is preferable that the next session be of type for which the number of pending requests in the token queue is the *maximum*. However, this simple approach may lead to starvation of a request. To avoid starvation, every pending request in the token queue is associated with an attribute called *age*; it measures the number of sessions that have been initiated since the request was added to the queue. Therefore, every time a new session is initiated, the age of all (pending) requests in the token queue increases by one.

Now, to select the type for the next session, all requests in the queue are grouped based on their type. Each type is then assigned a priority, which consists of two parts. The first part depends on the *group size*, which is given by

the number of requests in the group for that type. The second part depends on the *group age*, which is given by the sum of the ages of all the requests in the group for that type. The priority of a type is then computed by simply adding the two parts. The next session that is initiated is of type for which the priority value is the *maximum*. As explained earlier, one of the processes in the group is selected to become the next primary token holder and other processes in the group become secondary token holders.

We show that this approach prevents starvation, that is, it ensures that every request for critical section is eventually fulfilled.

3.3 Proof of Correctness

The following property about our algorithm can be easily verified:

Proposition 1 *There is exactly one primary token in the system at any time. Moreover, two tokens with the same session number are of the same type.*

The following lemma can be proved using induction on the session number:

Lemma 2 *A token belonging to session s with $s > 1$ is busy only if all tokens with session number $s - 1$ have been released.*

The safety property can now be proved using Proposition 1 and Lemma 2:

Theorem 3 (group mutual exclusion) *If two processes are concurrently executing their critical sections, then both critical sections are of the same type.*

To establish starvation freedom, we first show the following:

Lemma 4 *If there is a pending request of type that is different from the type of the current session, then the current session eventually terminates.*

For convenience, we refer to the act of selecting type for a session, which is done using a priority-based scheme, as a *step*. Note that the priority assigned to a type by our scheme is non-zero if and only if there is at least one pending request (for critical section) of that type. Further, if there are one or more pending requests for a type, the priority of the type strictly increases with each step until the type is "selected" for a session. In case the priority of a type increases, let l and u with $0 < l \leq u$ denote the lower and upper bounds, respectively, on the amount by which the priority can increase. For our scheme, $l = 1$ and $u = n$. Now, consider a type x whose current priority is greater than some value, say v. Observe that once y has been selected as a type for some session, at least $v/(u - l)$ steps are required before the priority of y can "catch up" with that of x. Therefore, once the priority of x becomes at least $m * (u - l)$, within m steps, the priority of x attains the largest value. Therefore, we have,

Theorem 5 (starvation freedom) *Every request for critical section is eventually fulfilled.*

Finally, it can be verified that:

Theorem 6 (concurrent entry) *Our algorithm satisfies the concurrent entry property.*

3.4 Performance Analysis

Our algorithm exchanges three kinds of messages, namely REQUEST, RELEASE and TOKEN. Clearly, for each request, there are at most $n-1$ REQUEST messages, at most $n-1$ RELEASE messages and at most 1 TOKEN message. Also, all messages except the TOKEN message are of size $O(1)$; the TOKEN message is of size $O(n)$. Therefore,

Theorem 7 (message complexity and overhead) *The worst-case message complexity of our algorithm is $2n - 1$ and its amortized message overhead is $O(1)$.*

Assume that the system is heavily loaded. Once the current session terminates, the next session starts as soon as the primary token holder of the next session has (1) received the token from the primary token holder of the current session, and (2) received RELEASE messages from all secondary token holders of the current session. Thus,

Theorem 8 (synchronization delay) *The synchronization delay of our algorithm is at most t.*

Now, assume that the system is lightly loaded. A process, on generating a request for critical section, can enter the critical section as soon as (1) its REQUEST message has been received by the current primary token holder (which in turn sends the token to it), and (2) it has received the token. Hence,

Theorem 9 (waiting time) *The waiting time of our algorithm is at most $2t$.*

Finally, it can be easily verified that:

Theorem 10 (maximum concurrency) *The maximum concurrency of our algorithm is n.*

4 Experimental Results

We experimentally compare the performance of our token-based algorithm with RA2, whose asymptotic performance is comparable to that of our algorithm in all complexity measures. We simulate the two algorithms using an event-based simulator on a Dell machine equipped with Pentium 4 processor operating at 2.00 GHz with a RAM of 512 MB. Our simulation has the following parameters. There are n processes and m groups. To model non-uniform group access, we assume that α% of the groups are requested β% of the times. 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

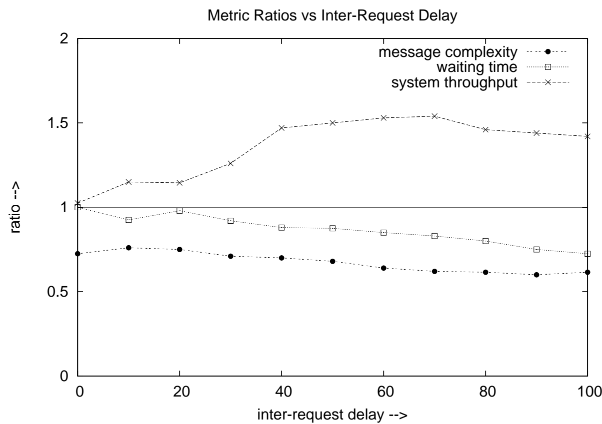

Figure 1. Performance of our algorithm relative to RA2.

distributed in the range $[0, 2 * \mu_{busy}]$ (with mean μ_{busy}). Message transmission delay is modeled to follow an exponential distribution with mean μ_{link}. In our simulation, we use the following values for various parameters: $n = 25$, $m = 50$, $\mu_{busy} = 2$ time units, $\mu_{link} = 4$ time units and bandwidth = 1000 integers per unit time. To simulate non-uniform request distribution, we assume that $\alpha = 20$ and $\beta = 80$. Each process generates 500 requests for critical section. All results are averaged over 10 runs.

We measure the performance of the two algorithms with respect to three metrics, namely message complexity, system throughput and waiting time. To make it easier to compare the two algorithms, we report the ratio (Our Algorithm/RA2) for the three metrics. Figure 1 depicts the variation in the ratios for the three metrics as the mean inter-request delay increases. As shown in the figure, our algorithm has *consistently* better performance than RA2 in all three metrics. Specifically, as the inter-request delay increases, message complexity decreases by as much as 40%, waiting time decreases by as much as 28% and system throughput increases by as much as 50%.

5 Conclusion and Future Work

In this paper, we proposed an efficient token-based distributed algorithm for solving the group mutual exclusion problem. Our algorithm is especially suited for applications in which a relatively small number of groups are requested more often than other groups. Our preliminary experiments indicate that our algorithm has much better performance than existing algorithms when group access is non-uniform.

Clearly, the scheme for selecting the type of the next session is crucial to the performance of our algorithm. As future work, we plan to investigate other type-selection schemes and measure their performance experimentally.

References

[1] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[2] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM (CACM)*, 24(1):9–17, January 1981.

[3] M. Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[4] I. Suzuki and T. Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.

[5] K. Raymond. A Tree based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[6] 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, October 1979.

[7] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, October 1971.

[8] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[9] Y.-J. Joung. Asynchronous Group Mutual Exclusion. *Distributed Computing (DC)*, 13(4):189–206, 2000.

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

[11] 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.

[12] 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.

[13] 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.

[14] Y.-J. Joung. Quorum-Based Algorithms for Group Mutual Exclusion. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(5):463–475, May 2003.

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

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

[17] P. K. Mohan. An Effi cient Distributed Group Mutual Exclusion Algorithm for Non-Uniform Group Access. Master's thesis, Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA, August 2005.