Quorum-Based Algorithms for Group Mutual Exclusion[*]

Yuh-Jzer Joung
Department of Information Management
National Taiwan University
Taipei, Taiwan

Keywords: mutual exclusion, group mutual exclusion, resource allocation, quorum system, surficial quorum system, coteries

Abstract

We propose a quorum system, which we referred to as the *surficial* quorum system, for group mutual exclusion. The surficial quorum system is geometrically evident and so is easy to construct. It also has a nice structure based on which a truly distributed algorithm for group mutual exclusion can be obtained, and processes' loads can be minimized. When used with Maekawa's algorithm, the surficial quorum system allows up to $\sqrt{\frac{2n}{m(m-1)}}$ processes to access a resource simultaneously, where n is the total number of processes, and m is the total number of groups. We also present two modifications of Maekawa's algorithm so that the number of processes that can access a resource at a time is not limited to the structure of the underlying quorum system, but to the number that the problem definition allows.

[*]Part of this research was done when the author was visiting Lab for Computer Science, Massachusetts Institute of Technology (1999-2000). Research supported in part by the National Science Council, Taipei, Taiwan, Grants NSC 89-2213-E-002-110, NSC 90-2213-E-002-026, and 37225F. A preliminary version of this paper appeared in *Proc. 15th International Symposium on DIStributed Computing (DISC 2001), Lecture Notes in Computer Science 2180*, pp. 16–32, October 2001.

1 Introduction

Mutual exclusion is one of the most fundamental problems in distributed systems. In this problem, access to a shared resource (i.e., entering a critical section) by concurrent processes must be synchronized so that only one process can use the resource at a time. Following Dijkstra's seminal paper [15], an extensive amount of research has been devoted to this subject in the past three decades or so. Several extensions to this problem have also been proposed: The *l-exclusion* problem [17] concerns situations in which at most l processes can be in the critical section simultaneously. The problems of *Dining Philosophers* [16], *Drinking Philosophers* [10], and *Committee Coordination* [11] concern situations in which a process needs to acquire more than one resource at a time and resources cannot be shared between processes.

Recently, there was another extension called *group mutual exclusion* [28], which concerns situations in which a resource can be shared by processes with a common property, e.g., they belong to the same "group", or they will not invoke "conflicting" operations while using the resource. Processes with different properties must use the resource in a mutually exclusive style. As an application of the problem, assume that data is stored in a CD jukebox where only one disk can be loaded for access at a time. Then when a disk is loaded, users that need data on this disk can concurrently access the disk, while users that need a different disk have to wait until no one is using the currently loaded disk. Although both group mutual exclusion and l-exclusion allow more than one process to be in the critical section simultaneously, the two problems differ in that in l-exclusion the conflict in accessing a resource is due to the number of processes that attempt to access the resource, while in the group mutual exclusion the conflict is due to the "type" of processes (i.e., the group they belong to).

Group mutual exclusion also plays a role in emerging wireless applications [26, 27, 6]. In these applications, two devices that wish to communicate with each other must be in the same frequency, and packets sent by a device in a frequency channel can be received by all devices listen to the same frequency within radio range of the sender. Thus, given a particular frequency channel, a group of devices that wish to communicate with one another must first acquire the channel before communication, and other groups of devices that wish to communicate must wait until the channel is cleared. Another example that deals with server cache can be found in [22].

Solutions for group mutual exclusion in shared memory models have been proposed in [28, 32, 3, 22, 44]. Here we consider message passing networks. The solutions in [45, 7, 8] all use a unique token circulating along a ring to resolve mutual exclusion among different groups of processes in accessing a shared resource (i.e., the critical section). The use of ring topology greatly simplifies the solutions, but it inevitably incurs long synchronization delay. The two message-passing solutions proposed in [30] are for general networks. Both are extensions of Ricart and Agrawala's algorithm for mutual exclusion [42]. Basically, they work as follows: a process wishing to enter a critical section broadcasts a request to all processes in the system, and enters the critical section when all processes have acknowledged its request. Thus the minimum synchronization delay for a process to enter a critical section is only two message transmission time. However, since all processes are involved in determining whether a process can enter a critical section, the algorithms cannot tolerate any single process failure. (Clearly, none of the aforementioned token-based algorithms can tolerate process failures, either.)

In the literature, quorum systems have proven useful in coping with site failures and network partitions for both mutual exclusion (e.g., [20, 37, 4, 1, 33, 14, 41]), and l-exclusion (e.g., [19, 31, 35, 39]). In general, a quorum system (called a *coterie* [20]) consists of a set of quora, each of which is a set of processes. Quora are used to guard critical sections. To enter a critical section, a process must *acquire* a quorum; that is, to obtain permission from every member of the quorum. Suppose that a quorum member gives permission to only one process at a time. Then mutual exclusion can be guaranteed by requiring every two quora in a coterie to intersect, and l-exclusion can be guaranteed by requiring any collection of $l+1$ quora in a coterie (called an *l-coterie* [19, 25]) to contain at least two intersecting quora. A quorum usually involves only a subset of the processes in the system. So even if processes may fail or become unreachable due to network partitions, some process may still be able to enter a critical section so long as not all quora are hit (a quorum is *hit* if some of its members fails).

It is easy to see that coteries for l-exclusion cannot be used for group mutual exclusion because two conflicting processes may then both enter a critical section after acquiring two disjoint quora respectively. On the other hand, coteries for mutual exclusion can be used for group mutual exclusion, but it will result in a degenerate solution in which only one process can be in a critical section at a time.

In this paper we present a quorum system, which we refer to as the *surficial* quorum system, for group mutual exclusion. To our knowledge, this is the first quorum system for group mutual exclusion to appear in the literature. The surficial quorum system is geometrically evident and so is easy to construct. It also has a nice structure based on which a truly distributed algorithm for group mutual exclusion can be obtained, and processes' loads can be minimized. When used with Maekawa's algorithm [37], the surficial quorum system allows up to $\sqrt{\frac{2n}{m(m-1)}}$ processes to access a resource simultaneously, where n is the total number of processes, and m is the total number of groups. In contrast, only one process is allowed to access a resource at a time if ordinary quorum systems are used.

We also present a modification of Maekawa's algorithm so that the number of processes that can access a resource at a time is not limited to the structure of the underlying quorum system, but to the number that the problem definition allows. Thus, the modified algorithm can also use ordinary quorum systems to solve group mutual exclusion. Nevertheless, when used with group quorum systems, the message complexity of the modified algorithm is still better than that used with an ordinary quorum system. Another modification that trades offs synchronization delay for message complexity is also presented in the paper.

The rest of the paper is organized as follows. Section 2 gives the problem definition. Section 3 presents the surficial quorum system. Section 4 presents quorum-based algorithms for group mutual exclusion. Conclusions and future work are offered in Section 5.

2 The Group Mutual Exclusion Problem

We consider a system of n asynchronous processes $1, \ldots, n$, each of which cycles through the following three states, with *NCS* being the initial state:

- *NCS*: the process is outside CS (the *Critical Section*), and does not wish to enter CS.
- *trying*: the process wishes to enter CS, but has not yet entered CS.
- *CS*: the process is in CS.

The processes belong to m groups $1, \ldots, m$. To make the problem more general, we do not require groups to be disjoint. When a process may belong to more than one group, the process must identify a unique group to which it belongs when it wishes to enter CS. Since group membership is concerned only at the time a process wishes to enter CS (and at the time the process is in CS), when we say 'process i belongs to group j', we implicitly assume that process i has specified j as its group for entering CS.[1]

The problem is to design an algorithm for the system satisfying the following requirements:

[1]The problem is described in a more anthropomorphous setting as *Congenial Talking Philosophers* in [28].

mutual exclusion: At any given time, no two processes of different groups are in CS simultaneously.

lockout freedom: A process wishing to enter CS will eventually succeed.

Moreover, to avoid degenerate solutions and unnecessary synchronization, we are looking for algorithms that can facilitate "**concurrent entering**", meaning that if a group g of processes wish to enter CS and no other group of processes are interested in entering CS, then the processes in group g can concurrently *enter* CS [28, 32, 22].

3 A Quorum System for Group Mutual Exclusion

In this section we present the surficial quorum system for group mutual exclusion. We begin with the definition of group quorum systems.

***Definition* 3.1** *Let $P = \{1, \ldots, n\}$ be a set of nodes.*[2] *An **m-group quorum system** $\mathfrak{C} = (C_1, \ldots, C_m)$ over P consists of m sets, where each $C_i \subseteq 2^P$ is a set of subsets of P satisfying the following properties:*

intersection: $\forall 1 \le i, j \le m, i \ne j, \forall Q_1 \in C_i, \forall Q_2 \in C_j : Q_1 \cap Q_2 \ne \emptyset$.

minimality: $\forall 1 \le i \le m, \forall Q_1, Q_2 \in C_i, Q_1 \ne Q_2 : Q_1 \not\subseteq Q_2$.

*We call each C_i a **cartel**, and each $Q \in C_i$ a **quorum**.*

Intuitively, $\mathfrak{C}$ can be used to solve group mutual exclusion as follows: each process i of group j, when attempting to enter CS, must acquire a quorum $Q \in C_j$ it has chosen by obtaining permission from every member of the quorum. Upon exiting CS, process i returns the permission to the members of the quorum. Suppose a quorum member gives permission to only one process at a time. Then, by the intersection property, no two processes of different groups can be in CS simultaneously. The minimality property is used rather to enhance efficiency. As is easy to see, if $Q_1 \subseteq Q_2$, then a process that can obtain permission from every member of Q_2 can also obtain permission from every member of Q_1. Note that the above concept of acquiring quora to enter CS is essentially from Maekawa's well-known algorithm [37] for standard mutual exclusion. We will discuss this algorithm in more detail in Section 4.

[2]The terms *processes* and *nodes* will be used interchangeably throughout the paper. For a distinguishing purpose, however, we use "nodes" specifically to denote quorum members, and "processes" to denote group members.

Recall that a quorum system over P for mutual exclusion is a set $C \subseteq 2^P$ of quora satisfying the following requirements:

intersection: $\forall Q_1, Q_2 \in C : Q_1 \cap Q_2 \neq \emptyset$.

minimality: $\forall Q_1, Q_2 \in C, Q_1 \neq Q_2 : Q_1 \not\subseteq Q_2$.

To distinguish quorum systems for mutual exclusion from group quorum systems, we refer to the former as ***ordinary*** quorum systems.

An ordinary quorum system C can be used as an m-group quorum system by a straightforward transformation $\mathfrak{T}_m$: $\mathfrak{T}_m(C) = (C, \ldots, C)$. By the intersection property of C, all quora in a cartel of $\mathfrak{T}_m(C)$ are pairwise intersected. Note that, in general, quora in the same cartel of a group quorum system need not intersect.

***Definition* 3.2** *The **degree** of a cartel C, denoted as $\mathsf{deg}(C)$, is the maximum number of pairwise disjoint quora in C. The **degree** of a group quorum system $\mathfrak{C} = (C_1, \ldots, C_m)$, denoted as $\mathsf{deg}(\mathfrak{C})$, is defined as $\mathsf{deg}(\mathfrak{C}) = \min\{\mathsf{deg}(C_i) \,|\, 1 \leq i \leq m\}$; that is, the minimum degree of the cartels. $\mathfrak{C}$ is of **uniform degree** k if all its cartels have the same degree k.*

Clearly, if a node gives out its permission to at most one process at a time (as in Maekawa's algorithm), then the number of processes of the same group that can be in CS simultaneously is limited to $\mathsf{deg}(C)$, where C is the cartel associated with the group. Moreover, a group quorum system of degree k immediately implies that every cartel contains at least an unhit quorum even if $k - 1$ processes have failed. So high degree group quorum systems also provide a better protection against faults. On the other hand, an m-group quorum system of degree k also implies that every quorum in the system has size at least k (unless $m = 1$). So the higher the degree, the larger the quorum size.

In the following we present an m-group quorum system $\mathfrak{S}_m = (C_1, \ldots, C_m)$ of uniform degree $\sqrt{\frac{2n}{m(m-1)}}$. In addition, the quora in the system satisfy the following four extra conditions:

1. $\forall 1 \leq i, j \leq m : |C_i| = |C_j|$.
2. $\forall 1 \leq i, j \leq m, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1| = |Q_2|$.
3. $\forall p, q \in P : |n_p| = |n_q|$, where n_p is the multiset $\{Q \,|\, \exists 1 \leq i \leq m : Q \in C_i \text{ and } p \in Q\}$, and similar for n_q. In other words, $|n_p|$ is the number of quora involving p.
4. $\forall 1 \leq i, j \leq m, i \neq j, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1 \cap Q_2| = 1$.

Intuitively, the first condition ensures that each group has an equal chance in competing for CS. The second condition ensures that the number of messages needed per entry to CS is independent of the quorum a process chooses. The third condition means that each node shares the same responsibility in the system. As argued by Maekawa [37], these three conditions are desirable for an algorithm to be truly distributed. The last condition simply minimizes the number of nodes that must be common to any two quora of different cartels, thereby reducing the size of a quorum.

Before presenting the detailed construction of $\mathfrak{S}_m$, we first provide some intuitions. It is easy to see that a 1-group quorum system $\mathfrak{S}_1$ satisfying the above conditions can be obtained as follows: $\mathfrak{S}_1 = \big(\{\{p\} \mid p \in P\}\big)$. The quorum system can be viewed as a line consisting of n nodes, each of which corresponds to a process in P, where $n = |P|$. Each quorum then consists of exactly one node on the line, and the collection of the quora constitutes the only cartel in the system. See Figure 1, top. By extending this line to a two-dimensional plane, we can obtain a 2-group quorum system $\mathfrak{S}_2 = (C_1, C_2)$, where each quorum in C_1 corresponds to the set of nodes in each row, while each quorum in C_2 corresponds to the set of nodes in each column. By taking one step further, we can construct a 3-group quorum system $\mathfrak{S}_3 = (C_1, C_2, C_3)$ by arranging nodes on the surface of a cube. Notice that a cube can be "wrapped up" by lines (strings) along three different dimensions. Lines along the same dimension are parallel to each other, while any two lines along different dimensions must intersect at two points. If we arrange the nodes on only three sides of the cube as shown in Figure 1, then every two lines along different dimensions intersect at exactly one point.

We can unfold the three sides of the cube on the plane as shown on the left of Figure 2. Each quorum in C_1 then corresponds to a vertical line across the first (right most) column of squares. Each quorum in C_2 corresponds to a horizontal line across the top square, and a vertical line across the left square on the bottom. Finally, each quorum in C_3 corresponds to a horizontal line across the two squares on the bottom.

By appending another three squares to the bottom of the above pile of squares and extending the lines to these extra squares, we can construct $\mathfrak{S}_4$ as shown on the right of Figure 2. Each quorum in C_1 corresponds to a vertical line across the first column of squares. Each quorum in C_2 corresponds to a horizontal line across the square on the first level (starting from the top), and then a vertical line across the second column of squares. Each quorum in C_3 corresponds to a horizontal line across the squares on the second level, and then a vertical line across the third column of squares. Finally, each quorum in C_4 corresponds to a horizontal line across the squares on the third level. Notice that on the right staircase of Figure 2, the first level of squares constitutes

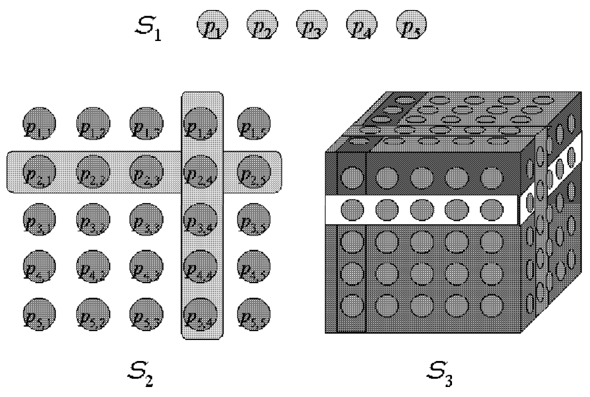

Figure 1: The surficial quorum system for $m = 1, 2$, and 3.

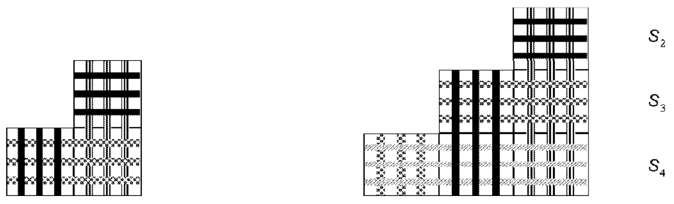

Figure 2: The unfolded surficial quorum system $\mathfrak{S}_3$ (left), and $\mathfrak{S}_4$ (right).

$\mathfrak{S}_2$, and the first two levels of squares constitutes $\mathfrak{S}_3$.

This procedure can be extended to $\mathfrak{S}_m$. In general, each C_i in $\mathfrak{S}_m$ needs $m - 1$ squares, each of which is to be shared with one of the other $m - 1$ cartels so that the corresponding lines of the two cartels intersect at exactly one node on the square. Overall, there are $m(m-1)/2$ squares. Let k be the width of each square (i.e., the number of quora in each cartel). Then each square consists of k^2 nodes. So the total number of nodes on the $m(m-1)/2$ squares is $k^2m(m-1)/2$. A simple way to map nodes on the squares to the processes in P is to let each node correspond to a unique process. In this case $k^2m(m-1)/2 = n$, where $n = |P|$. So $k = \sqrt{\frac{2n}{m(m-1)}}$, $m > 1$. So the quorum size is $(m-1) \cdot k = \sqrt{\frac{2n(m-1)}{m}}$ and each cartel consists of $\sqrt{\frac{2n}{m(m-1)}}$ quora.

Figure 4 presents the "staircase" construction of the surficial quorum system. We shall use $\mathfrak{F}$ to denote the construction. When given input P and integer m as specified in Figure 4, $\mathfrak{F}(P, m)$ denotes the result $\mathfrak{S}_m$. The following theorem summarizes some properties of the surficial quorum system.

Theorem 3.1 *Let P be an n-set and m be a nonnegative integer such that $m > 1$ and $\sqrt{\frac{2n}{m(m-1)}} = k$ for some integer k. Furthermore, let $\mathfrak{F}(P, m) = (C_1, \ldots, C_m)$. Then $\mathfrak{F}(P, m)$ is an m-group quorum*

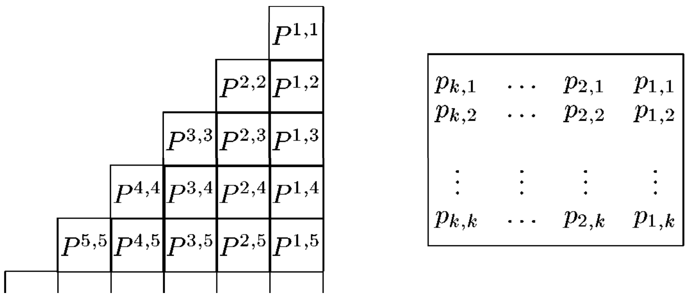
$P^{1,1}$
$P^{2,2}$ $P^{1,2}$
$P^{3,3}$ $P^{2,3}$ $P^{1,3}$
$P^{4,4}$ $P^{3,4}$ $P^{2,4}$ $P^{1,4}$
$P^{5,5}$ $P^{4,5}$ $P^{3,5}$ $P^{2,5}$ $P^{1,5}$
$p_{k,1}$ $\cdots$ $p_{2,1}$ $p_{1,1}$
$p_{k,2}$ $\cdots$ $p_{2,2}$ $p_{1,2}$
$p_{k,k}$ $\cdots$ $p_{2,k}$ $p_{1,k}$

Figure 3: Arrangement of nodes in $\mathfrak{S}_m$. On the left is the indices (superscripts) of squares, and on the right is the indices (subscripts) of nodes in each square.

Input. An n-set P and a nonnegative integer m.

Output. An m-group quorum system $\mathfrak{S}_m = (C_1, \ldots, C_m)$ over P.

Assumption. $m > 1$ and $\sqrt{\frac{2n}{m(m-1)}} = k$ for some integer k.

1. Partition P into $\frac{m(m-1)}{2}$ subsets, each of which consists of k^2 nodes. Let $P^{i,j}$ denote these subsets, $1 \leq i \leq m-1$, $i \leq j \leq m-1$.
   For each $P^{i,j}$, let $p^{i,j}_{r,s}$ denote the nodes in the set, where $1 \leq r, s \leq k$. (See Figure 3.)
2. For each cartel C_i in $\mathfrak{S}_m$, denote the quora in the cartel by $Q_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq k$. Then, $Q_{i,j}$ is defined by
$$Q_{i,j} = \left\{ p^{s,i-1}_{r,j} \,\middle|\, 1 \leq s \leq i-1,\ 1 \leq r \leq k \right\} \bigcup \left\{ p^{i,s}_{j,r} \,\middle|\, i \leq s \leq m-1,\ 1 \leq r \leq k \right\}$$
   Note that when $i = 1$, the first set in the formula is empty, and when $i = m$, the second part is empty.

Figure 4: The staircase construction $\mathfrak{S}$.

system over P, and is of uniform degree $\sqrt{\frac{2n}{m(m-1)}}$. Moreover, $\mathfrak{S}(P,m)$ satisfies the following conditions:

- $\forall 1 \leq i, j \leq m : |C_i| = |C_j| = \sqrt{\frac{2n}{m(m-1)}}$.
- $\forall 1 \leq i, j \leq m, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1| = |Q_2| = \sqrt{\frac{2n(m-1)}{m}}$.
- $\forall p, q \in P : |n_p| = |n_q| = 2$, *where n_p is the multiset $\{Q \mid \exists 1 \leq i \leq m : Q \in C_i \text{ and } p \in Q\}$, and similar for n_q.*
- $\forall 1 \leq i, j \leq m, i \neq j, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1 \cap Q_2| = 1$.

Proof. Straightforward from the construction. □

Below we provide some comments on the construction. First, we provide an upper bound on the degree of group quorum systems.

Lemma 3.2 *Let $\mathfrak{C}$ be an m-group quorum system over an n-set. If $m > 1$, then $\mathfrak{C}$ has degree $k \le \sqrt{n}$.*

Proof. This follows from the fact that if $m > 1$ and $\mathsf{deg}(\mathfrak{C}) = k$, then by the intersection property every quorum in a cartel must have at least k nodes. Since there are at least k pairwise disjoint sets in a cartel, the total number of different nodes involved in a cartel is at least k^2, which must be less than n. So $k \le \sqrt{n}$. □

In fact, the above bound is also tight as we can construct an m-group quorum system with (uniform) degree $\sqrt{n}$ [29]. So, in terms of degree, the construction of the surficial quorum system $\mathfrak{S}_m$ (which has uniform degree $\sqrt{\frac{2n}{m(m-1)}}$) is not optimal. However, to reach the optimal degree, n must be equal to some p^{2c}, where p is a prime and c is a positive integer. Besides, the construction is difficult to visualize as it works on an affine plane of order p^c. In contrast, the surficial quorum system is easy to visualize and so is easy to construct.

Moreover, observe that in the surficial quorum system every node is involved in at most two quora. Clearly, a node's load can be minimized by letting it be included in at most two quora. (If some node is included in only one quorum, then the node is redundant as removing it from the quorum does not affect the intersection and minimality properties.) As shown in the following lemma, if every node is included in at most two quora, then the maximum degree an m-group quorum system can achieve is $\sqrt{\frac{2n}{m(m-1)}}$, which is exactly what $\mathfrak{S}_m$ has achieved. So the surficial quorum system achieves maximal degree when processes' loads are minimized.

Lemma 3.3 *Let $\mathfrak{C} = (C_1, \ldots, C_m)$ be an m-group quorum system over an n-set P, $m > 1$. Define n_p, $p \in P$, to be the multiset $\{Q \mid \exists 1 \le i \le m : Q \in C_i \text{ and } p \in Q\}$. If $\forall p \in P : |n_p| \le 2$, then $\mathfrak{C}$ has degree $k \le \sqrt{\frac{2n}{m(m-1)}}$.*

Proof. Let $P^{i,j}_{r,s}$ be the intersection of the rth quorum in C_i and the sth quorum in C_j, $i \ne j$. By definition, $P^{i,j}_{r,s} \ne \emptyset$. Since every node is included in at most two quora, $P^{i,j}_{r,s} \cap P^{i',j'}_{r',s'} = \emptyset$ if $\{i, j\} \ne \{i', j'\}$ (i.e., the two *unordered* pairs are not equal) or $(r, s) \ne (r', s')$ (i.e., the two *ordered* pairs are not equal). Given that $1 \le i \ne j \le m$ and that each cartel contains at least k quora, there are $\binom{m}{2}$ unordered pairs of i, j, and for each of them, at least k^2 ordered pairs of (r, s) that can constitute a $P^{i,j}_{r,s}$. So $\left|\bigcup_{i,j,r,s} P^{i,j}_{r,s}\right| \ge \binom{m}{2}k^2$. Since $\binom{m}{2}k^2 \le n$, we have $k \le \sqrt{\frac{2n}{m(m-1)}}$. □

In the construction of the surficial quorum system we have assumed a one-to-one mapping from nodes to processes. One can experiment different mappings as well as adjust the dimension of the squares to tune the quorum system to fit into different applications.

Garcia-Molina and Barbara [20] proposed the notion of *dominance* to compare the failure resilience of ordinary quorum systems. A quorum system C *dominates* D if $\forall\, Q \in D, \exists\, R \in C : R \subseteq Q$. Intuitively, this means that whenever a quorum in D can survive some failures, then some quorum in C can certainly survive as well. Thus in this sense C is said to be superior to D because C provides more protection against failures. Similarly, dominance of group quorum systems can be defined as follows.

***Definition* 3.3** *Let $\mathfrak{C} = (C_1, \ldots, C_m)$ and $\mathfrak{D} = (D_1, \ldots, D_m)$ be two m-group quorum systems over P. Then $\mathfrak{C}$ dominates $\mathfrak{D}$ if*

1. $\mathfrak{C} \neq \mathfrak{D}$,

2. $\forall\, Q \in D_i, 1 \leq i \leq m, \exists\, R \in C_i : R \subseteq Q$.

*$\mathfrak{D}$ is **nondominated** if there is no $\mathfrak{C}$ such that $\mathfrak{C}$ dominates $\mathfrak{D}$.*

Theorem 3.4 *Let $\mathfrak{F}(P, m) = (C_1, \ldots, C_m)$ be the group quorum system constructed in Figure 4. Then $\mathfrak{F}(P, m)$ is dominated.*

Proof. In Figure 4, let $T = Q_{1,1} - \{p^{1,1}_{1,1}\} \cup \{p^{1,1}_{2,1}\}$. Then T intersects $Q_{i,j}$ for all $1 < i \leq m$, $1 \leq j \leq k$. Moreover, $T \not\subseteq Q_{1,j}$ and $Q_{1,j} \not\subseteq T$. So $(C_1 \cup \{T\}, C_2, \ldots, C_m)$ is an m-group quorum system and it dominates $\mathfrak{F}(P, m)$. □

"Fully distributedness" and "nondominance" appear to be two conflicting notions, as for example, the "fully distributed" ordinary quorum system proposed by Maekawa [37] is also dominated [18, 40]. However, the construction of $\mathfrak{S}_m = (C_1, \ldots, C_m)$ is such that every quorum Q in the cartels is a minimal set that intersects every other quorum in a different cartel. As proved in [29], this property implies that $\mathfrak{S}_m$ can be enlarged to a nondominated group quorum system $\mathfrak{U} = (D_1, \ldots, D_m)$ such that $C_i \subseteq D_i$ for all $1 \leq i \leq m$. In other words, we can build upon $\mathfrak{S}_m$ a nondominated group quorum system $\mathfrak{U}$ such that $\mathfrak{U}$ "contains" $\mathfrak{S}_m$. An important meaning of this "containing" relation is that: $\mathfrak{S}_m$ can be used to realize a truly distributed algorithm when failures do not occur, while $\mathfrak{U}$ can be used to "backup" $\mathfrak{S}_m$ when failures do occur to increase fault tolerance. Therefore, based on $\mathfrak{S}_m$ we can easily construct a nondominated group quorum system to support a fully distributed algorithm for group mutual exclusion.

4 Quorum-Based Algorithms for Group Mutual Exclusion

In this section we present algorithms that use quorum systems to solve group mutual exclusion. The network is assumed to be complete and reliable.

4.1 The Basic Framework: Maekawa's Algorithm

Most quorum-based algorithms for mutual exclusion are based on Maekawa's algorithm [37], which works as follows:

1. A process p wishing to enter CS chooses a quorum Q, and it must "lock" all nodes of Q before it can enter CS. It does so by sending a lock request REQUEST to each node of Q.

   When p has locked all nodes of Q, it enters CS. Upon leaving CS, i sends an UNLOCK message to all nodes of Q to release their locks.

2. A node can be locked by one process at a time. So upon receipt of a request by p, a node i checks if it is locked.

   (a) If i is locked by another process q, then some arbitration mechanism is used to determine whether to let p wait, or to let p preempt q's possession of i's lock.

   (b) Otherwise, i sends a lock (a message LOCKED) to p, and is now "locked by p."

In general, lock requests by p to the nodes of Q are sent simultaneously (i.e., multicast) so as to minimize the *synchronization delay*, which is the delay from the time a process invokes a mutual exclusion request to the time it enters CS. The delay is measured in terms of message transmission time. It is clear that the minimum synchronization delay is 2 if lock requests are multicast. However, due to the asynchrony of the system, a process may hold a lock while waiting for another process to release a lock. This in turn may incur a deadlock.

Maekawa's algorithm handles deadlocks by requiring low-priority processes to yield locks to high-priority processes. Priorities are usually implemented by Lamport's logical timestamps [36]. The smaller the timestamp of a lock request, the higher the priority of the request. Specifically, if a node i receives a lock request by p after giving a lock to q and p's priority is higher than q's priority, then i issues an inquiry message to q. Process q then returns i's lock (by sending an UNLOCK message) if it cannot successfully lock all members of the quorum it chooses. Node i then gives its lock to p after receiving the lock from q. When p exits CS and releases i's lock, i returns the lock to q (presumably no other process with priority higher than q is also waiting for i's lock). So

Maekawa's algorithm needs $3c$ to $6c$ messages per entry to CS, where c is the (maximum) size of a quorum.

Maekawa's algorithm has been well studied in [37, 43, 12, 13, 9]. It is easy to see that the algorithm can be directly adapted to group mutual exclusion as follows: Let $\mathfrak{C} = (C_1, \ldots, C_m)$ be an m-group quorum system over P. A process $p \in P$ that wishes to enter CS as a member of group g chooses a quorum from the cartel C_g, and enters CS only when it has locked all members of the quorum. By the mutual exclusion property of $\mathfrak{C}$ and by the conflict resolution scheme used in the algorithm, mutual exclusion and lockout freedom are easily guaranteed.

4.2 A Tradeoff Between Concurrency and Message Complexity

In Maekawa's algorithm, since a node can be locked by only one process at a time, the maximum number of processes of a group that can be in CS simultaneously is limited to the degree of the cartel associated with the group. So no concurrency is offered using group quorum systems $\mathfrak{T}_m(C)$ derived from ordinary quorum systems C (because $\mathfrak{T}_m(C)$ has degree one).

For group quorum systems with degree greater than one, they may still not be able to offer a satisfactory degree of concurrency. This is because the size of a group can be greater than $\sqrt{|P|}$. However, as discussed in Section 3, no m-group quorum system over P can have degree more than $\sqrt{|P|}$, unless $m = 1$.

To overcome this, we modify Maekawa's algorithm to allow a node to be locked by more than one process. So even if quora in the same cartel may intersect, two or more processes may still enter CS simultaneously. The new rule for a node i to determine whether to grant p's lock request is as follows:

> Node i grants p's request so long as there is no *conflict*—i.e., no other process of a different group has also requested i's lock. Otherwise, conflicts are resolved as follows: group g yields i's locks to group h if there is a process in group h that has priority higher than all processes of group g that have requested i's locks.

Because of the conflict resolution scheme, after a node i is locked by l processes, i may have to retrieve all its locks if it receives a higher priority request from a different group. Retrieving a lock from a process results in three messages: an inquiry message, an unlock message, and eventually the return of the lock to the process. So $3l$ messages may be generated to resolve a conflict. On the other hand, l also relates to the number of processes that may be in CS simultaneously. So in our algorithm we shall limit the number of locks a node may give away at a time, and use it as a parameter to control the message complexity while allowing maximum concurrency. Later in Section 4.2.1 we discuss how to set this parameter for a given quorum system.

For ease of understanding, the detailed code of the algorithm is presented as two CSP-like repetitive commands consisting of guarded commands [24]: Figure 5 describes the behavior of a process that acts as a group member, and Figure 6 describes the behavior of a node that acts as a quorum member. If one wishes, the two repetitive commands can be combined into a single one. We refer to the algorithm as Maekawa_M.

A repetitive command in CSP is of the form

$$*[gc_1 \,\Box\, gc_2 \,\Box\, \ldots \,\Box\, gc_k]$$

Each gc_i is called a *guarded command*, which is of the form

$$b; \textbf{receive}\ msg \longrightarrow S$$

where b is a boolean condition called the *boolean* guard, and "**receive** *msg*" is called the *message* guard. Both guards are optional. A guarded command can be executed only if it is *enabled*, i.e., its boolean guard evaluates to true and the specified message in the message guard has arrived. The execution receives the message and then the command S (which may consist of a sequence of statements) is executed atomically. If there is more than one enabled command, then one of them is chosen for execution, and the choice is nondeterministic. We do, however, require that a guarded command that is continuously enabled be executed eventually.

In the algorithm, priorities are represented by pairs $\langle p, t \rangle$, where p is a process/node id and t is a timestamp. Priorities are compared as follows:

$$\langle p, t_1 \rangle > \langle q, t_2 \rangle \text{ iff } t_1 > t_2 \text{ or } t_1 = t_2 \wedge p > q$$

The larger the value, the lower the priority.

Furthermore, we assume that the communication channel between each pair of processes is FIFO. The FIFO assumption simplifies the design of the algorithm; otherwise, care must be taken to deal with this out-of-order message transmission.

4.2.1 Analysis of Maekawa_M

In this section we prove the correctness of Maekawa_M and analyze its complexity. **Invariant**, **unless**, and **leads-to** assertions [11] are used in reasoning about the algorithm. We use $p.v$ to

```
A.1  *[ wish to enter CS as a member of group $g \longrightarrow$
A.2         $group := g$;
A.3         $state := trying$;
A.4         select an arbitrary quorum $Q$ from $cartel(g)$;
A.5         $sn := sn + 1$;
A.6         $priority := \langle p, sn \rangle$;
A.7         multicast REQUEST($priority, g$) to every node in $Q$; /* issue a lock request */

B.1  □  receive INQUIRE($i, v, t$) $\longrightarrow$
B.2         $sn := \max(sn, t)$;
B.3         if $v = priority \bigwedge i \in locked\_nodes \bigwedge state \neq CS$ then [
B.4             send UNLOCK($p, sn, false$) to $i$; /* unlock $i$ */
B.5             $locked\_nodes := locked\_nodes - \{i\}$; ]

C.1  □  receive LOCKED($i, t$) $\longrightarrow$
C.2         $sn := \max(sn, t)$;
C.3         $locked\_nodes := locked\_nodes \cup \{i\}$;
C.4         if $|locked\_nodes| = Q$ then $state := CS$; /* enter CS */

D.1  □  exit CS $\longrightarrow$
D.2         $state := NCS$;
D.3         for $j \in locked\_nodes$ do send UNLOCK($p, sn, true$) to $j$;
D.4         $locked\_nodes := \emptyset$;
D.5         $priority := \langle p, \infty \rangle$;
D.6  ]
```

Variables:

- *state*: the state of process p.
- *group*: the group to which p belongs.
- *sn*: a sequence number used to implement p's logical clock. It is initialized to 0.
- *priority*: the priority of p's request. It is initialized to a maximal value $\langle p, \infty \rangle$.
- Q: the quorum p selects.
- *cartel*(g): the cartel associated with group g.
- *locked_nodes*: set of nodes p has locked. It is initialized to $\emptyset$.

Messages:

- REQUEST(v, g): a lock request by p of group g. v is the priority of the request.
- INQUIRE(i, v, t): a message by node i to ask the recipient to release i's lock. v is the priority of the recipient's lock request. It is used by the recipient to determine if the message is out-of-date; that is, for a previous request made by the recipient. t is the timestamp of the message.
- UNLOCK(p, t, *complete_flag*): a message by p to unlock the recipient. t is the timestamp of the message. *complete_flag* is a flag indicating if p unlocks the recipient because it has completed its lock request (i.e., has already entered CS).
- LOCKED(i, t): a "lock" given by node i. t is the timestamp of the message.

Figure 5: Algorithm Maekawa_M executed by process p.

```
E.1   *[ receive REQUEST($\langle p,t\rangle, g$) $\longrightarrow$
E.2         $sn := \max(sn, t)$;
E.3         $requests := requests \cup \{(p,t,g)\}$;
E.4         if $lock\_ps = \emptyset \bigvee (lock\_group = g \bigwedge has\_priority)$ then [
E.5             if $|lock\_ps| < \max[g]$ then [
E.6                 send LOCKED($i, sn$) to $p$;
E.7                 $lock\_ps := lock\_ps \cup \{p\}$;
E.8                 if $|lock\_ps| = 1$ then [
E.9                     $lock\_group := g$;
E.10                    $has\_priority := true$; ] ]
E.11            else [ /* a maximum number of locks have been given to group $g$ */
E.12                $(q_1, t_1, h_1) := rank(\max[g], g, P, requests)$; /* get the lowest priority request that
                        should be granted */
E.13                $(q_2, t_2, h_2) := rank(-1, g, lock\_ps - inquired\_ps, requests)$; /* get the lowest
                        priority request that is granted but not yet inquired */
E.14                if $\langle q_2, t_2\rangle > \langle q_1, t_1\rangle$ then [ /* $q_2$ has to yield $i$'s lock */
E.15                    $inquired\_ps := inquired\_ps \cup \{q_2\}$;
E.16                    send INQUIRE($i, \langle q_2, t_2\rangle, sn$) to $q_2$; ] ]
E.17        else if $g \neq lock\_group \bigwedge has\_priority$ then [ /* check if $lock\_group$ still has the priority */
E.18            $(q_1, t_1, h_1) := rank(1, lock\_group, P, requests)$;
E.19            if $\langle p, t\rangle < \langle q_1, t_1\rangle$ then [ /* current $lock\_group$ loses the priority */
E.20                $has\_priority := false$;
E.21                for $(q, t', h) \in requests$ s.t. $q \in lock\_ps - inquired\_ps$ do [ /* retrieve locks */
E.22                    send INQUIRE($i, \langle q, t'\rangle, sn$) to $q$;
E.23                    $inquired\_ps := inquired\_ps \cup \{q\}$; ] ]
E.24        /* else the request is queued */

F.1   □ receive UNLOCK($p, t, flag$) $\longrightarrow$
F.2         $sn := \max(sn, t)$;
F.3         $lock\_ps := lock\_ps - \{p\}$;
F.4         $inquired\_ps := inquired\_ps - \{p\}$;
F.5         if $flag = true$ then /* $p$ has entered CS and so delete its request */
F.6             $requests := requests - \{(p, -, -)\}$;
F.7         $(q_1, t_1, h_1) := rank(1, -, P, requests)$; /* get the highest priority request in queue */
F.8         if $has\_priority \bigwedge lock\_ps \neq \emptyset \bigwedge lock\_group \neq h_1$ then [ /* $lock\_group$ loses the priority */
F.9             $has\_priority := false$;
F.10            for $(q, t', h) \in requests$ s.t. $q \in lock\_ps - inquired\_ps$ do [ /* retrieve locks */
F.11                send INQUIRE($i, \langle q, t'\rangle, sn$) to $q$;
F.12                $inquired\_ps := inquired\_ps \cup \{q\}$; ] ]
```

Figure 6: Algorithm Maekawa_M executed by node i.

```
F.13        else if $has\_priority \bigwedge lock\_group = h_1$ then [
F.14            $(q, t', h) := rank(1, lock\_group, P - lock\_ps, requests)$;
F.15            if $q \neq \perp$ then /* another group member is waiting for $i$'s lock */
F.16                send LOCKED$(i, sn)$ to $q$;
F.17                $lock\_ps := lock\_ps \cup \{q\}$; ]
F.18        else if $lock\_ps = \emptyset \bigwedge requests \neq \emptyset$ then [
F.19            $has\_priority := true$;
F.20            $lock\_group := h_1$;
F.21            for $k := 1$ to max$[h_1]$ do [
F.22                $(q, t', h) := rank(k, h_1, P, requests)$;
F.23                if $q \neq \perp$ then [
F.24                    send LOCKED$(i, sn)$ to $q$;
F.25                    $lock\_ps := lock\_ps \cup \{q\}$; ] ] ]
F.26  ]
```

Variables:

- *requests*: set of lock requests received by i. The requests are represented as tuples (p, t, g), where $\langle p, t\rangle$ is the priority of the request, and g is the group of the requester. The set is initialized to $\emptyset$.

  When referring to tuples, we sometimes use $(p, -, -)$ to denote the tuple whose second and third fields are don't-care. Given an integer k, a group h, and a set of processes $Q \subset P$, we assume the following function on *requests*:

  - $rank(k, h, Q, requests)$: return the kth highest priority (or $|k|$th lowest priority if $k < 0$) request in $\{(p, t, g) \mid (p, t, g) \in requests \wedge g = h \wedge p \in Q\}$ if it exists; otherwise return $(\perp, \perp, \perp)$. If the input parameter h is given as '$-$', then the corresponding condition '$g = h$' for the parameter is omitted. So $rank(k, -, Q, requests)$ returns the kth highest priority request in $\{(p, t, g) \mid (p, t, g) \in requests \wedge p \in Q\}$.

- *lock_ps*: set of processes to which i has given a lock. It is initialized to $\emptyset$.
- *inquired_ps*: set of processes to which i has sent an inquiry message, but i has not yet received their unlock messages. It is initialized to $\emptyset$.
- *sn*: a sequence number used to implement i's logical clock. It is initialized to 0.
- *lock_group*: the group of the processes to which i has given a lock. It is initialized to $\perp$.
- *has_priority*: a Boolean variable indicating if *lock_group* still has the priority to enter CS. It is initialized to *false*.
- max$[g]$: maximum number of locks node i can give to group g.

Figure 6: Algorithm Maekawa_M executed by node i (cont.).

represent p's local variable v. Also, $p.requests(q)$ denotes q's request in $p.requests$ (if it exists), and $p.requests(q, priority)$ denotes the priority of the request, and $p.requests(q, group)$ denotes the group id specified in the request. When reading the proof, keep in mind that each guarded command in the algorithm is executed as an atomic unit. This greatly simplifies the reasoning. The reasoning uses the following predicates:

- $in\_CS(p) \triangleq p.state = \mathrm{CS}$
- $request(p,i) \triangleq \exists\, t, g : (p,t,g) \in i.requests$

The following assertions can be easily proved from the algorithm, and so we omit the details. Note that unless stated otherwise, variables p, q are universally quantified over processes, and i, j are universally quantified over nodes.

(I1) **invariant** $in\_CS(p) \Rightarrow p.Q \in cartel(p.group) \;\wedge\; \forall\, i \in p.Q : i \in p.locked\_nodes$

(I2) **invariant** $i \in p.locked\_nodes \;\wedge\; i \in q.locked\_nodes \Rightarrow p.group = q.group$

(I3) **invariant** $request(p,i) \Rightarrow i.lock\_ps \neq \emptyset$

(I4) **invariant** $request(p,i) \;\wedge\; p \notin i.lock\_ps \Rightarrow$

$$\begin{aligned}
&\big(i.lock\_group = i.requests(p, group) \;\wedge\; |i.lock\_ps| = \max[i.lock\_group] \\
&\quad \wedge\; \forall\, q \in i.lock\_ps : i.requests(q, priority) < i.requests(p, priority)\big) \\
\vee\; &\big(i.lock\_group = i.requests(p, group) \;\wedge\; |i.lock\_ps| = \max[i.lock\_group] \\
&\wedge\; \exists\, q \in i.lock\_ps : i.requests(q, priority) > i.requests(p, priority) \;\wedge\; q \in i.inquired\_ps\big) \\
\vee\; &\big(i.lock\_group \neq i.requests(p, group) \;\wedge \\
&\quad \exists\, q \in P : request(q,i) \;\wedge\; i.requests(q, priority) < i.requests(p, priority)\big) \\
\vee\; &\big(\forall\, q \in i.lock\_ps : q \in i.inquired\_ps\big)
\end{aligned}$$

Theorem 4.1 Maekawa_M *guarantees mutual exclusion.*

Proof. If two processes p and q from different groups are both in CS, then by the mutual exclusion property of group quorum systems, the quora they choose must involve a common node, say i. By (I1), both p and q must lock i, and by (I2), p and q must belong to the same group; contradiction. □

The following lemmas are needed for proving lockout freedom.

Lemma 4.2 $request(p, i) \wedge \forall q \in P - \{p\} : request(q, i) \Rightarrow i.requests(p, priority) < i.requests(q, priority)$ **leads-to** $(p \in i.lock\_ps \wedge p \notin i.inquired\_ps) \vee (\exists r \in P : request(r, i) \wedge i.requests(r, priority) < i.requests(p, priority))$.

Proof. Suppose p has requested a lock from node i. Suppose further that p's request has priority higher than all requests to i, and remains to be the highest priority among all the requests that may arrive at i (or, otherwise, the lemma is vacuously proven). By (I3) and (I4), either (1) i will grant p a lock, (2) $p.group = i.lock\_group$ and i has inquired one process in $i.lock\_ps$, or (3) i has inquired all the processes in $i.lock\_ps$. In Case (2), i will eventually retrieve a lock and then give it to p (see guarded command (F) of Maekawa_M). In Case (3), i will eventually retrieve all its locks and then give it to p (and the processes of the same group that have requested lock of i). So in any of the three cases, i eventually gives p a lock. Certainly, when i gives p the lock, it will not inquire p to return the lock at the same time. The lemma therefore follows. □

Lemma 4.3 $p \notin i.inquired\_ps$ **unless** $\exists q \in P : request(q, i) \wedge i.requests(q, priority) < i.requests(p, priority)$.

Proof. This follows directly from the fact that i will not inquire p to retrieve its lock unless a higher priority process is wishing to acquire i's lock. □

Theorem 4.4 Maekawa_M *guarantees lockout freedom.*

Proof. By Lemmas 4.2 and 4.3, when a process wishes to enter CS and chooses a quorum to acquire, if its request has priority higher than all existing requests, then it will eventually acquire the quorum and enter CS. Priorities are implemented by logical timestamps that are nondecreasing, and a process increases its logical clock each time it initiates a request for CS. So after p multicasts a request REQUEST$(\langle p, sn\rangle, g)$ to the members of a quorum, the number of requests with priorities higher than $\langle p, sn\rangle$ that could occur in the system is bounded.[3] Because a process spends only finite time in CS, the requests with priorities higher than $\langle p, sn\rangle$ will eventually cease to exist. So p eventually acquires its quorum and enters CS. □

For message complexity of Maekawa_M, recall that after a node i has given locks to l processes, a new lock request may cause i to withdraw all its locks. Withdrawing a lock from a process

[3]The use of logical timestamps further ensures that after a node i receives p's request, each other process can have at most one request at node i with priority higher than $\langle p, sn\rangle$, thereby reducing p's waiting time to enter CS.

results in three messages: an inquiry message, an unlock message, and eventually the return of the lock to the process. So in the worst case, a lock request by p to i may generate $3l$ messages, in addition to i's LOCKED message to p and p's UNLOCK message to i. So the message complexity of Maekawa_M is as follows:

Theorem 4.5 *Let c be the maximum quorum size in the group quorum system associated with* Maekawa_M*, and g be the group such that $\forall 1 \leq h \leq m : \max[g] \geq \max[h]$. Then, in* Maekawa_M *the worst case number of messages a process may generate in order to enter* CS *is* $3c + 3c \cdot \max[g]$.

Theorem 4.6 *In* Maekawa_M*, the minimum synchronization delay for a process to enter* CS *is* 2 *message transmission time.*

Proof. This is because lock requests are multicast. So in the best case, two message transmission time (one for a lock request and the other for the grant of the request) is enough for a process to enter CS. □

To facilitate a maximum concurrency while minimizing message complexity, for each group g with cartel C, we can partition g into $\mathsf{deg}(C)$ subgroups, and assign $\mathsf{deg}(C)$ pairwise disjoint quora in C to them, one quorum per subgroup. In a regular use of the quorum system (where failures do not occur), each process selects the quorum assigned to its subgroup when the process wishes to enter CS. In this case, each subgroup consists of $s/\mathsf{deg}(C)$ members, where s is the size of g. So each node needs only to give away at most $s/\mathsf{deg}(C)$ locks at a time, and the algorithm still allows all s members of the group to be in CS simultaneously. So an entry to CS requires at most $3c + 3c \cdot s/\mathsf{deg}(C)$.

For example, suppose the surficial quorum system $\mathfrak{S}(P, m)$ presented in Section 3 is used in Maekawa_M, and suppose each group size s equals to n, i.e., every process may potentially enter CS as a member of any group. Then the message complexity is

$$(1) \quad 3\sqrt{\frac{2n(m-1)}{m}} + 3\sqrt{\frac{2n(m-1)}{m}} \cdot \frac{n}{\sqrt{\frac{2n}{m(m-1)}}} = O(n \cdot m)$$

In contrast, if we do not bound the number of locks a node may give away at a time, then a node may give away $n-1$ locks before it encounters a conflict. So the message complexity becomes

$$(2) \quad 3\sqrt{\frac{2n(m-1)}{m}} + 3\sqrt{\frac{2n(m-1)}{m}} \cdot n = O(\sqrt{n} \cdot n)$$

Note that in the construction of $\mathfrak{F}(P, m)$, m is at most $O(\sqrt{n})$. So Equation (1) is $O(n \cdot \min\{m, \sqrt{n}\})$. For applications in which $m << \sqrt{n}$ (e.g., m is some constant that is independent of n), Equation (1) shows some advantage in bounding the number of locks a node may give away at a time. Bounding the number of locks is also useful in applications in which the number of processes that can be concurrently in CS should be restricted, (e.g., to guarantee quality of service).

4.3 A Tradeoff Between Concurrency and Synchronization Delay

If message complexity needs to be bounded in $O(c)$, deadlocks must be resolved in a different way. A well-known technique in resource allocation is to let each process lock quorum members in some fixed order [23], say, with increasing node IDs. So if a process p is waiting for locking i, then every lock p holds must be from some j such that $j < i$. Moreover, every process q that currently locks i has either locked all members of its quorum, or is waiting for locking some k such that $k > i$. So deadlocks are not possible because there is no circular waiting.

Note that the above deadlock free argument does not depend on how many locks a node may give out at a time. That is, a node can still be locked by multiple processes. The number of messages required per entry to CS is $3c$, and the minimum synchronization delay is $2c$. The message complexity and the minimum synchronization delay can be reduced further to $2c + 1$ and $c + 1$, respectively, by letting quorum members circulate request messages. The complete code of the algorithm is given in Figures 7 and 8. We refer to the algorithm as Maekawa_S. Note that auxiliary variables in the code are used only to assist the analysis (see Section 4.3.1).

In the algorithm a node i may receive a lock request by p before it receives p's unlock message for p's previous request (lines D.10-11), regardless of whether communication channels are FIFO or not. This is because request messages hop through quorum members. So when p issues an unlock message msg_1 to i and then issues a new request msg_2, msg_2 may arrive at i (indirectly through members of a different quorum) before msg_1 does. To simplify the algorithm, i defers the process of msg_2 (which is called "early request" in the algorithm) until it has received msg_1 (lines E.17-19).

Like Maekawa_M, requests are not processed strictly in FCFS order. Instead, when a node i receives a lock request from a process p of group g, if i has no outstanding lock, then i grants p's request and chooses p as a *reference* (line D.5). A reference process is used such that subsequent lock requests from the same group are also granted until the reference process exits CS and unlocks the node. When a reference process unlocks i, if no other process of a different group is waiting

```
A.1  *[ wish to enter CS as a member of group $g \longrightarrow$
A.2      $state := trying$;
A.3      $group := g$;
A.4      select an arbitrary quorum $Q$ from $cartel(g)$;
A.5      **send** REQUEST$(p, g, Q, sn)$ **to** $first(Q)$;

B.1  □ **receive** GRANT$(i)$ $\longrightarrow$ /* acquired quorum $Q$ */
B.2      $locked\_nodes := Q$; /* auxiliary variable */
B.3      $state := CS$;

C.1  □ exit CS $\longrightarrow$
C.2      $sn := sn + 1$; /* auxiliary variable */
C.3      $state := NCS$;
C.4      **for** $i \in Q$ **do send** UNLOCK$(p)$ **to** $i$;
C.5      $locked\_nodes := \emptyset$; /* auxiliary variable */
C.6  ]
```

Variables:

- *state*: the state of process p.
- *sn*: an auxiliary variable, initialized to 1, that is used to number p's lock requests. Thus, $sn - 1$ denotes the number of entries to CS p has made.
- *group*: the group to which p belongs.
- Q: the quorum p selects. We assume the following three functions on quora:
  - $first(Q)$: return the smallest ID in Q.
  - $next(k, Q)$: return the smallest ID in Q that is greater than k.
  - $last(Q)$: return the largest ID in Q.
- $cartel(g)$: the cartel associated with group g.
- *locked_nodes*: an auxiliary variable that denotes the set of nodes p has locked. It is initialized to $\emptyset$.

Messages:

- REQUEST(p, g, Q, v): a lock request by p, of group g, to lock the nodes in Q. v is the sequence number of the request. Note that v is used only to assist the proof.
- GRANT(i): a message from i (which must be the last node in Q) notifying p that it has locked all nodes in Q.
- UNLOCK(p): a message by p to release the recipient's lock.

Figure 7: Algorithm Maekawa_S executed by process p.

```
D.1  *[ receive REQUEST($p, g, Q, sn$) $\longrightarrow$
D.2      if $lock\_ps = \emptyset \bigvee (lock\_group = g \bigwedge reference \neq \perp \bigwedge p \notin lock\_ps)$ then [
           /* grant the request */
D.3        if $lock\_ps = \emptyset$ then [
D.4          $lock\_group := g$;
D.5          $reference := p$; ]
D.6        if $i = last(Q)$ then send GRANT($i$) to $p$;
D.7        else send REQUEST($p, g, Q, sn$) to $next(i, Q)$;
D.8        $lock\_ps := lock\_ps \cup \{p\}$;
D.9        $requests := requests \cup \{(p, g, Q, sn)\}$; ] /* auxiliary variable */
D.10     else if $p \in lock\_ps$ /* $p$'s request arrives before its previous unlock message */
D.11       $early\_requests := early\_requests \cup \{(p, g, Q, sn)\}$;
D.12     else [ $deferred\_req := deferred\_req \cup \{(p, g, Q, sn)\}$;
D.13         $requests := requests \cup \{(p, g, Q, sn)\}$; /* auxiliary variable */ ]

E.1  □ receive UNLOCK($p$) $\longrightarrow$
E.2      $lock\_ps := lock\_ps - \{p\}$;
E.3      $requests := requests - \{(p, -, -, -)\}$; /* auxiliary variable */
E.4      if $reference = p$ then
E.5        if $lock\_ps \neq \emptyset \wedge deferred\_req = \emptyset$ then /* choose a new reference */
E.6          $reference := q$ for some arbitrary $q \in lock\_ps$;
E.7        else $reference := \perp$;
E.8      if $lock\_ps = \emptyset \bigwedge deferred\_req \neq \emptyset$ then [ /* grant requests from the earliest group */
E.9        $(q, h, R, sn) := first(deferred\_req)$;
E.10       $reference := q$;
E.11       $lock\_group := h$;
E.12       for $(q', h', R', sn') \in deferred\_req$, $h = h'$, do [
E.13         if $i = last(R')$ then send GRANT($i$) to $q'$;
E.14           else send REQUEST($q', h', R', sn'$) to $next(i, R')$;
E.15         $deferred\_req := deferred\_req - \{(q', h', R', sn')\}$;
E.16         $lock\_ps := lock\_ps \cup \{q'\}$; ]
E.17     if $(p, g, S, v) \in early\_requests$ for some $g, S$, and $v$ then [ /* process $p$'s early request */
E.18       send REQUEST($p, g, S, v$) to $i$;
E.19       $early\_requests := early\_requests - \{(p, g, S, v)\}$; ]
E.20 ]
```

Figure 8: Algorithm Maekawa_S executed by node i.

Variables:

- *deferred_req*: queue of lock requests that are deferred by i. The requests are represented as quadruples (p, g, Q, sn), where p is the requester, g is the group of the requester, Q is the quorum i chooses, and sn is the sequence number of the request (and note that the sequence number is used only to assist the proof). The queue is initialized to $\emptyset$. Requests in the queue are ordered by the time they are inserted into the queue. Function *first*(*deferred_req*) returns the earliest request in the queue.
- *requests*: an auxiliary variable, initialized to $\emptyset$, that denotes the set of lock requests received by i. Note that the set does not include "early" lock requests (see below).
- *early_requests*: queue of "early" lock requests received by i. A request by p to i is said "early" and is temporarily stored in *early_requests* if the request arrives at i before p's previous release message.
- *lock_ps*: set of processes to which i has given a lock. It is initialized to $\emptyset$.
- *lock_group*: the group of the processes to which i has given a lock. It is initialized to $\perp$.
- *reference*: a reference process used to determine whether subsequent processes of *lock_group* can enter CS.

Figure 8: Algorithm Maekawa_S executed by node i (cont.).

for i's lock, then a new reference is chosen from those processes that currently lock i (lines E.4-7). Otherwise, the reference is reset to $\perp$, meaning that the "door" to CS (guarded by node i) for the group is closed to yield the opportunity to another group.

An obvious reason for choosing this "entry policy" is to increase resource utilization. Clearly, by the mutual exclusion property, while some reference process p is in CS, no other group of processes can be in CS. So maximal resource utilization can be achieved by allowing more processes of the same group to share CS with p, regardless of whether some other group of processes are waiting for CS or not. Furthermore, because while p is in CS, some fast process may enter and exit CS any number of times, the algorithm facilitates an unbounded degree of concurrency [28]. Note that lockout freedom can still be guaranteed because a reference process will eventually exit CS and close the "door" to CS for its group.

Another reason for choosing this "entry policy" is to minimize the number of "context switches", and to reduce the overall average delay in waiting for CS. (A *context switch* occurs when the next entry to CS is by a process of a different group [28].) As analyzed in [30], in group mutual exclusion requests to CS cannot be processed in a strictly FCFS order, or else the system could degenerate to the case in which nearly only one process can be in CS at a time when m is large. This would then result in a large number of context switches and long waiting time. Several entry policies have been studied in [30]. In general, a good entry policy must allow late processes to "jump over" processes that have been waiting for CS so that the late processes can share CS concurrently with other processes of the same group that are already in CS. So in both Maekawa_M and Maekawa_S we allow some late requests to overtake existing requests to acquire a node's lock.

4.3.1 Analysis of Maekawa_S

In this section we prove the correctness of Maekawa_S and analyze its complexity. Like for Maekawa_M, we first establish some assertions for Maekawa_S. The predicates $\mathit{in\_CS}(p)$ and $\mathit{request}(p, i)$ are defined as in Section 4.2.1. Also, $i.\mathit{requests}(p, sn)$ denotes the sequence number of p's lock request in $i.\mathit{requests}$. In the algorithm, $p.sn$ is an auxiliary variable used to number p's lock requests, and $p.sn - 1$ denotes the number of entries to CS p has made. So $p.sn \geq i.\mathit{requests}(p, sn)$ all the time. When $p.sn > i.\mathit{requests}(p, sn)$, p's request at i must be obsolete; that is, p has returned i's lock (after exiting CS) but the return message has not yet arrived at i.

(I5) **invariant** $\mathit{in\_CS}(p) \Rightarrow p.Q \in \mathit{cartel}(p.\mathit{group}) \ \wedge\ \forall i \in p.Q : i \in p.\mathit{locked\_nodes}$

(I6) **invariant** $i \in p.\mathit{locked\_nodes} \ \wedge\ i \in q.\mathit{locked\_nodes} \Rightarrow p.\mathit{group} = q.\mathit{group}$

(I7) **invariant** $request(p, i) \Rightarrow i.lock\_ps \neq \emptyset$
(I8) **invariant** $p \in i.lock\_ps \Rightarrow request(p, i)$

The following invariant follows from the fact that a process acquires nodes' locks in the order of increasing node IDs.

(I9) **invariant**$p \in i.lock\_ps \ \wedge \ i.requests(p, sn) = p.sn \Rightarrow \forall j \in p.Q :$

$$j > i \ \vee \ p \in j.lock\_ps$$

Theorem 4.7 Maekawa_S *guarantees mutual exclusion.*

Proof. Proof similar to Theorem 4.1. □

Lemma 4.8 *request*(p, i) **leads-to** $p \in i.lock\_ps$.

Proof. Without loss of generality, assume that $P = \{1, 2, \ldots, n\}$. We prove the lemma by induction on i in decreasing order.

For the induction basis (i.e, $i = n$), assume that $request(p, n)$ holds but $p \notin n.lock\_ps$. By (I7), n must have given its lock to some process q. By (I8) and (I9), either q has already entered CS and has sent an UNLOCK message to return i's lock, or all nodes in $q.Q$ has granted q's lock requests. In the former case, n will eventually receive the UNLOCK message and delete q from $n.lock\_ps$. In the latter case, q will eventually make an entry to CS and return n's lock. So if $p \notin n.lock\_ps$, then every process in $n.lock\_ps$ will eventually return n's lock and be removed from $n.lock\_ps$. Recall that a process q whose request arrives at n later than p's request can overtake p to acquire n's lock only if there is a granted request in $n.requests$ such that the request arrives at n earlier than p's request does. So eventually no request can overtake p's request. So eventually n will grant p's request.

For the induction step, assume that $request(p, i)$ holds but $p \notin i.lock\_ps$ for some $i < n$. Again, by (I7), i must have given its lock to some process q. So either (1) all nodes in $q.Q$ has granted q their locks, or (2) some node $j \in q.Q$ has not yet granted q's lock request. In the latter case, by (I8) and (I9), $j > i$. By the induction hypothesis, j eventually grants q's request. So all nodes in $q.Q$ will eventually grant q's lock request. So in either case, q eventually acquires the locks of $q.Q$ and enters CS. After exiting CS q will return i's lock. So while $p \notin i.lock\_ps$, every process in $i.lock\_ps$ will eventually return i's lock and be removed from $i.lock\_ps$. Although a process after returning i's lock may issue a new request and overtake p to acquire i's lock, by an argument similar to the induction basis, we can show that eventually no process can overtake p to acquire i's lock. The lemma is therefore established. □

Theorem 4.9 Maekawa_S *guarantees lockout freedom.*

Proof. When a process p wishes to enter CS, it issues a lock request to *first*$(p.Q)$. When the request arrives, if *first*$(p.Q)$ still keeps p's old request, then the new request will be placed in *early_requests* until p's UNLOCK message for the old request arrives at *first*$(p.Q)$. Because message transmission time is finite, p's (new) request eventually leads to *request*(p, i). Then by Lemma 4.8 i eventually grants p's request and forwards the request to the next node in $p.Q$. Similarly, the next node eventually grants p's request and forwards it to the next node, and so on until every node in $p.Q$ has granted p's request. Then p can enter CS, thereby guaranteeing lockout freedom. □

The following complexity follows directly from the algorithm.

Theorem 4.10 *In* Maekawa_S, *it takes* $2c + 1$ *messages for a process to enter* CS, *where* c *is the size of the quorum the process chooses. Moreover, the minimum synchronization delay for the process to enter* CS *is* $c + 1$ *message transmission time.*

4.4 Remarks: Group Quorum Systems vs. Ordinary Quorum Systems

One may have observed that ordinary quorum systems can also be used in Maekawa_M and Maekawa_S to solve group mutual exclusion. A natural question then is whether group quorum systems are beneficial over ordinary quorum systems. To address this, let k be the group size (which is the number of members that may simultaneously access the CS). There are two cases to consider:

1. k is small enough (i.e., $k \leq \sqrt{n}$) that k-degree group quorum systems exist.
2. k is larger than $\sqrt{n}$, and hence, multiple locks must be accommodated.

When Maekawa_M is used, in Case 1, the worst case message complexity of an ordinary quorum system Q is $O(c(Q) \cdot k)$, where $c(Q)$ is the largest size of a quorum in Q. A k-degree group quorum system Q' has message complexity $O(c(Q'))$, but we also have the constraint $c(Q') \geq k$.

In Case 2, the worst case message complexity of a regular quorum system Q is again $O(c(Q) \cdot k)$. For example, the FPP quorum system in [37] (which also supports a truly distributed solution) has $O(\sqrt{n} \cdot k)$ complexity. On the other hand, a $\sqrt{n}$-degree group quorum system Q' has message complexity $O(c(Q') \cdot k/\sqrt{n})$. For example, the affine plane group quorum system in [29] has $O(k)$ complexity.

If applications can tolerate long synchronization delay, then Maekawa_S can be used. In this case, there is not much difference in choosing between group quorum systems and ordinary quorum systems. However, ordinary quorum systems have been extensively studied in the literature, and they have been optimized in many possible ways. For example, the majority quorum systems have the highest availability [40] when failure probability $p < 1/2$, while the triangle lattices presented in [5] have optimal load and optimal cost of failures. So ordinary quorum systems offer a wide variety of choices in the design. Group quorum systems, on the other hand, are new, and so many of their properties remain to be explored. Note also that group quorum systems with degree higher than 1, in general, come with a cost in *availability* [40].

5 Conclusions and Future Work

We have presented a quorum system, the surficial quorum system, for group mutual exclusion. The surficial quorum system generalizes existing quorum systems for mutual exclusion in that quora for processes of the same group need not intersect with one another. This generalization allows processes to acquire quora simultaneously, and so allows them to enter critical section concurrently. The surficial quorum system has a very simple geometrical structure, based on which a truly distributed algorithm for group mutual exclusion can be obtained, and based on which processes' loads can be minimized.

The surficial quorum system has degree $\sqrt{\frac{2n}{m(m-1)}}$, where n is the total number of processes, and m is the total number of groups. So when used with Maekawa's algorithm, it allows a maximum of $\sqrt{\frac{2n}{m(m-1)}}$ processes to be in the critical section simultaneously. The message complexity per entry to the critical section is $O(\sqrt{\frac{2n(m-1)}{m}})$. Furthermore, it can tolerate up to $\sqrt{\frac{2n}{m(m-1)}} - 1$ process failures. For comparison, the two message-passing algorithms RA1 and RA2 presented in [30] have message complexity $2n$ and $3n$, respectively, and both allow all group members to be in the critical section simultaneously. However, they cannot tolerate any single process failure. In terms of minimum synchronization delay, all three algorithms have the same measure—2.

As we have noted earlier, the degree of group quorum systems is theoretically bounded by $\sqrt{n}$. So Maekawa's algorithm must be generalized if group size is greater than $\sqrt{n}$ and we wish to allow all group members to be in the critical section simultaneously. Two generalizations Maekawa_M and Maekawa_S were presented in the paper. Both allow all group members to be in the critical section simultaneously, regardless of the degree of the underlying quorum systems. Maekawa_M preserves Maekawa's minimum synchronization delay, but needs $O(c \cdot s/d)$ messages per entry to the critical section, where c is the quorum size, s is the group size, and d is the degree of the underlying group quorum system. The other algorithm Maekawa_S reduces the message complexity to $2c + 1$, but needs a minimum synchronization delay of $c + 1$.

There is a considerable literature on ordinary quorum systems. Many structures have been explored, including *finite projective planes* [37], *weighted voting* [21, 20, 2], *grids* [14, 34], *trees* [1], *wheels* [38], *walls* [41], and *planar graphs* [5]. The surficial quorum system can be viewed as the counterpart of grids in the group mutual exclusion version, while the affine planes proposed in [29] are the counterpart of finite projective planes. For future work, it is interesting to investigate how the other structures can be used for group quorum systems.

Acknowledgments. I would like to thank Nancy Lynch for giving me a valuable opportunity to visit her group, and to discuss this research with her, as well as with the other members of the TDS group, including Idit Keidar, Alex Shvartsman, and Victor Luchangco. I would also like to thank Michel Raynal for stimulating this research, and the anonymous referees for their helpful comments.

References

[1] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.

[2] Mustaque Ahamad and Mostafa H. Ammar. Multidimensional voting. *ACM Transactions on Computer Systems*, 9(4):399–431, November 1991.

[3] K. Alagarsamy and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. Technical report, Dept. of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, 1999.

[4] Daniel Barbara and Hector Garcia-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1:119–132, 1986.

[5] Rida A. Bazzi. Planar quorums. *Theoretical Computer Science*, 243(1–2):243–268, July 2000.

[6] Bluetooth SIG. Http://www.bluetooth.com/.

[7] Sebastien Cantarell, Ajoy K. Datta, Franck Petit, and Vincent Villain. Group mutual exclusion in token rings. In *Proceedings of the 8th Colloquium on Structural Information and Communication Complexity (SIROCCO 2001)*, Vall de Nuria, Catalonia, Spain, June 27-29 2001. Carleton Scientific.

[8] Sebastien Cantarell, Ajoy K. Datta, Franck Petit, and Vincent Villain. Token based group mutual exclusion for asynchronous rings. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 691–694. IEEE Computer Society Press, 2001.

[9] Guohong Cao and Mukesh Singhal. A delay-optimal quorum-based mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1256–1268, December 2001.

[10] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[11] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[12] Ye-In Chang. A correct $O(\sqrt{N})$ distributed mutual exclusion algorithm. In *Proc. of the 5th International Conference on Parallel and Distributed Computing and Systems*, pages 56–61, October 1992.

[13] Ye-In Chang. Notes on Maekawa's $O(\sqrt{N})$ distributed mutual exclusion algorithm. In *Symposium on Parallel and Distributed Systems (SPDP '93)*, pages 352–359, Los Alamitos, Ca., USA, December 1994. IEEE Computer Society Press.

[14] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–59, December 1992.

[15] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[16] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, October 1971.

[17] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *20th Annual Symposium on Foundations of Computer Science*, pages 234–254, San Juan, Puerto Rico, 29–31 October 1979. IEEE.

[18] Wai Chee Ada Fu. *Enhancing concurrency and availability for database systems.* PhD thesis, Simon Fraser University, Burnaby, British Columbia, Canada, 1990.

[19] Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. Distributed k-mutual exclusion problem and k-coteries. In *Proceedings of the 2nd International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Computer Science 557*, pages 22–31, 1991.

[20] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[21] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Annual ACM Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, Cali., December 1979. ACM Press.

[22] Vassos Hadzilacos. A note on group mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Newport, Rhode Island, August 2001. ACM Press.

[23] J. W. Havender. Avoiding deadlock in multiasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[24] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[25] S.-T Huang, J.-R. Jiang, and Y.-C. Kuo. K-coteries for fault tolerant K entries to a critical section. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 74–81, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

[26] IEEE 802.11 Working Group for Wireless Local Area Networks. Http://ieee802.org/11/.

[27] IEEE 802.15 Working Group for Wireless Personal Area Networks (WPANs). Http://ieee802.org/15/.

[28] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, Puerto Vallarta, Mexico, June 1998. ACM Press. Full paper in *Distributed Computing*, 13(4):189-206, 2000.

[29] Yuh-Jzer Joung. On generalized quorum systems. Technical report, Department of Information Management, National Taiwan University, Taipei, Taiwan, 2000. Submitted for publication.

[30] Yuh-Jzer Joung. The congenial talking philosophers problem in computer networks. *Distributed Computing*, 15(3):155–175, 2002.

[31] Hirotsugu Kakugawa, Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. Availability of k-coterie. *IEEE Transactions on Computers*, 42(5):553–558, May 1993.

[32] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32. ACM Press, 1999.

[33] Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[34] Akhil Kumar, Michael Rabinovich, and Rakesh K. Sinha. A performance study of general grid structures for replicated data. In *13th International Conference on Distributed Computing Systems*, pages 178–185. IEEE Computer Society Press, 1993.

[35] Yu-Chen Kuo and Shing-Tsaan Huang. A simple scheme to construct k-coteries with $O(\sqrt{N})$ uniform quorum sizes. *Information Processing Letters*, 59(1):31–36, 8 July 1996.

[36] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[37] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[38] Yosi Marcus and David Peleg. Construction methods for quorum systems. Technical Report CS92-33, The Weizmann Institute of Science, Rehovot, Israel, 1992.

[39] Mitchell L. Neilsen. Properties of nondominated K-coteries. *The Journal of Systems and Software*, 37(1):91–96, April 1997.

[40] David Peleg and Avishai Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, December 1995.

[41] David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997.

[42] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[43] Mukesh Singhal. A class of deadlock-free Maekawa-type algorithms for mutual exclusion in distributed systems. *Distributed Computing*, 4(3):131–138, 1991.

[44] K. Vidyasankar. A simple group l-exclusion algorithm. *Information Processing Letters*, 85(2):79–85, November 2002.

[45] Kuen-Pin Wu and Yuh-Jzer Joung. Asynchronous group mutual exclusion in ring networks. *IEE Proceedings–Computers and Digital Techniques*, 147(1):1–8, 2000.

Yuh-Jzer Joung received his B.S. in electrical engineering from the National Taiwan University in 1984, and his M.S. and Ph.D. in computer science from the State University of New York at Stony Brook in 1988 and 1992, respectively. From 1999 to 2000, he was a visiting scientist at the Lab for Computer Science, Massachusetts Institute of Technology. He is currently a professor and the chair in the Department of Information Management at the National Taiwan University, where he has been a faculty member since 1992. His main research interests are in the area of distributed computing, with specific interests in multiparty interaction, fairness, (group) mutual exclusion, and peer-to-peer computing.