A quorum-based extended group mutual exclusion algorithm without unnecessary blocking

Yoshifumi Manabe
NTT Cyber Space Laboratories
Nippon Telegraph and Telephone Corporation
1-1 Hikarinooka, Yokosuka, 239-0847 Japan
manabe.yoshifumi@lab.ntt.co.jp

JaeHyrk Park*
IRIS
Information and Communications University
Yusong-ku, Taejon, 305-732 Korea
eye2174@icu.ac.kr

Abstract

This paper presents a quorum-based distributed algorithm for the extended group mutual exclusion problem. In the group mutual exclusion problem, multiple processes can enter a critical section simultaneously if they belong to the same group. Processes in different groups cannot enter a critical section at the same time. In the extended group mutual exclusion, each process is a member of multiple groups at the same time. Each process can select which group it belongs at making a request. The algorithm for the group mutual exclusion cannot be applied for this extended problem, since there can be a case that two processes are prevented from entering a critical section simultaneously even when they are capable of doing so. We call the above situation unnecessary blocking. We present a quorum-based algorithm that prevents unnecessary blocking and show its correctness proof.

1 Introduction

Mutual exclusion is a fundamental problem in distributed systems. When some resource (for example, a file, a communication channel, a printer) is shared among processes, no two processes are allowed to enter a critical section (CS) and use it at the same time. Many distributed mutual exclusion algorithms have been proposed [17][18]. A quorum-based algorithm [4] is a solution for fault-tolerance. Some extensions of the quorum-based algorithms have been discussed. In one there are multiple units of the same resource [9][14][15] and in another the set of resources that each process can access differs from process to process [10][19].

Recently, group mutual exclusion [7] has been proposed. There are multiple groups of processes. The processes in the same group can enter CS at the same time. This problem corresponds to the following situation. There is a CD jukebox and each process wants to read some data on the CDs. If CD A is loaded, multiple processes which want to read data on CD A can access it at the same time. These processes are in the same group. By contrast, the processes which want to read data on CD B cannot do so when A is loaded. These processes form a different group. In the original definition, each process is a member of just one group at each instant[1]. In [8], the following extended definition has been introduced. Some processes might be members of multiple groups at the same time. In the CD jukebox example, the same data might be copied CD B and C. In this situation, the user can read the data if B or C is currently loaded. This paper discusses this extended group mutual exclusion problem.

For group mutual exclusion, shared memory system algorithms [5][6][12][20], token-based algorithms [2][3][21] quorum-based algorithms[1][8][11] have been proposed. Though [8] discuss this extended group mutual exclusion problem, it just notes that when a process belongs to multiple groups, it arbitrary selects one group.

The above algorithm is not sufficient, since when some process p is entering CS, another process p', which can enter CS at the time, might be blocked. We call above situation as an unnecessary blocking. We show a new quorum-based algorithm which prevents unnecessary blocking.

Section 2 defines the extended group mutual exclusion problem. Section 3 briefly describes the former algorithm and its unnecessary blocking. Section 4 presents our new algorithm. Section 5 concludes the paper.

*Part of this work was undertaken while the author was staying at NTT Communication Science Laboratories.

[1]This definition allows that when a process makes a new request after using CD A, the new request might be for CD B.

2 Extended group mutual exclusion

A distributed system consists of processes and channels. The processes are asynchronous and fail in accordance with the fail-stop model. The process failure can be detected. Processes communicate with each other by passing messages through first-in, first-out (FIFO), asynchronous, and reliable (no message loss occurs) channels.

This paper assumes that the processes are divided into requesting processes and manager processes to simplify the discussion. The manager processes manages mutual exclusion and the requesting processes just make requests to enter CS. In actual systems, one process can perform both roles simultaneously. The discussion in this paper can also be applied to such systems. Let us denote $U = \{q_1, q_2, \ldots, q_n\}$ as the set of manager processes and $V = \{p_1, p_2, \ldots, p_l\}$ as the set of requesting processes.

$\mathcal{G} = \{g_1, g_2, \ldots, g_m\}$ is the set of groups. $g_i \subseteq V$ and $g_i \neq \phi$ for every $i(1 \leq i \leq m)$. The processes in g_i can enter CS at the same time. The set of groups p_i belongs to is called p_i's group set and is denoted as $G(p_i)$. That is, $G(p_i) = \{g \in \mathcal{G} | p_i \in g\}$. Each requesting process belongs to at least one group in $\mathcal{G}$, thus, $G(p_i) \neq \phi$.

In accordance with the example of the CD jukebox, the following is assumed as regards group selection [8]. When process p_i enters CS, it selects one group $g \in G(p_i)$, which corresponds to the selection of a CD. This group is called p_i's group selection and is denoted as $gs(p_i)$. The definition of extended group mutual exclusion is as follows.

mutual exclusion: Process p_i and p_j cannot be in CS at the same time if $gs(p_i) \neq gs(p_j)$. ■

starvation freedom: Every process that wants to enter CS must be eventually able to do so. ■

Now we define the unnecessary blocking freeness. A natural definition might be "when process p_i is in CS, any process p_j such that $gs(p_i) \in G(p_j)$ can enter CS at the same time," but this definition leads the following starvation scenario. Process $p_i(i = 1, 2, \ldots)$ such that $G(p_i) = \{g_1\}(i = 1, 2, \ldots)$ and p_0 such that $G(p_0) = \{g_0\}$ exist. Initially, p_1 makes a request and enters CS. p_0 then makes a request but it cannot enter CS. After that, p_2 makes a request. Because of the above unnecessary blocking freedom, existence of p_1 allows p_2 to enter CS. Then p_1 exits, but p_0 cannot enter CS because p_2 is currently entering CS. Then p_3 makes a request and existence of p_2 allows p_3 to enter CS, p_2 exits, p_4 makes a request and so on, and p_0 cannot enter CS forever.

The chain of "allowing another process to enter CS at the same time" leads a starvation. Thus, we prohibit a chain of allowance. We call p is a pivot process if no other process is in CS when p enters CS.

Unnecessary blocking freedom: When a pivot process p_i is entering CS, any process p_j such that $gs(p_i) \in G(p_j)$ can enter CS at the same time. ■

In the above scenario, p_1 is a pivot process. Thus, p_2 can enter CS at the same time. But p_2 is not a pivot process, thus, p_3 cannot enter CS even if p_2 is currently entering CS. After exiting of p_1, no new process enters CS until every currently entering process exits. Thus, p_0 eventually enters CS.

3 Problems in former algorithm

This section provides the outline of the algorithm in [8] and its unnecessary blocking. The m-group quorum system $\mathcal{C} = \{C_1, \ldots, C_m\}$ is defined as follows. C_i is the set of quorums for group g_i. Each quorum $Q \in C_i$ satisfies $Q \subseteq U$ and $Q \neq \phi$. m-group quorum system satisfies the following two properties.

Intersection property: $\forall Q_1 \in C_i$ and $\forall Q_2 \in C_j$, $Q_1 \cap Q_2 \neq \phi(1 \leq i, j \leq m, i \neq j)$.

Minimality property: $\forall Q, Q' \in C_i, Q \not\subseteq Q'(1 \leq i \leq m)$. ■

Each manager process has one "OK" vote to send. It can send "OK" to requesting processes in at most one group at the same time. The intersection property means that for any two requesting processes in different groups, the quorums intersects, thus these two processes cannot enter CS at the same time, because of the above manager processes' rule as regards sending "OK". The outline of their group mutual exclusion algorithm is as follows.

(when requesting process p wants to enter CS)

(1) p selects one group g_i from $G(p)$, selects one quorum Q from C_i, and sends "$Request(g_i)$" to every member of Q.

(2) When p receives "OK" from every member in Q, p enters CS. ■

(when manager process q receives "$Request(g_i)$" from p) q sends "OK" to p if

(1) q sends no "OK" to any other processes, or

(2) q has sent "OK" to a request p' such that $gs(p') = g_i$. ■

The actual algorithm is more complex to avoid starvation and achieve efficiency.

The above algorithm has two problems which lead to unnecessary blocking. First, consider the following example. p_1, whose group set $G(p_1)$ is $\{g_1\}$, sends "$Request(g_1)$" to every member in $Q \in C_1$. It receives "OK" from every member in $Q \in C_1$ and enters CS. p_2, whose group set $G(p_2)$ is $\{g_1, g_2\}$, then appears. The algorithm requires p_2 to select one group from $G(p_2)$ before making a request. Suppose that p_2 selects g_2 as $gs(p_2)$. Then, p_2 cannot enter CS because $gs(p_2) \neq g_1$. This blocking is unnecessary because p_2 could enter CS if p_2 would set g_1 as $gs(p_2)$.

This unnecessary blocking comes from the condition that p_2 must set $gs(p_2)$ when there is no information about the other requests. If p_2 can set $gs(p_2)$ after the current status is obtained (for example, some process whose group selection is g_1 is currently entering CS), p_2 can set a better group as $gs(p_2)$ and this type of unnecessary blocking is avoided.

For the second type of unnecessary blocking, consider the following example. There are two groups, g_1 and g_2. The quorum set for g_1 is $\{\{q_1, q_2\}, \{q_3, q_4\}\}$ and the quorum set for g_2 is $\{\{q_1, q_3\}, \{q_2, q_4\}\}$. These quorum sets satisfy the condition of a 2-group quorum system. Now suppose that there are three requests, p_1, p_2, and p_3. Assume that priorities of these requests satisfy $p_1 > p_2 > p_3$ ($x > y$ means x's priority is higher than that of y). $G(p_1) = G(p_3) = \{g_1\}$ and $G(p_2) = \{g_2\}$. Thus, $gs(p_1) = gs(p_3) = g_1$ and $gs(p_2) = g_2$. Note that $|G(p_i)| = 1$ for all i, which means that each request's group selection is unique. Thus, the following scenario is not affected by the above group selection problem, that is, the following problem exists even for group mutual exclusion. p_1 selects $Q_1 = \{q_3, q_4\}$ as its quorum, sends a request, receives "OK", and enters CS. Next, p_2 selects $\{q_1, q_3\}$ and sends a request. q_1 sends "OK" to p_2. However, since q_3 has sent "OK" to p_1, it does not reply "OK" to p_2. Lastly p_3 selects $Q_3 = \{q_1, q_2\}$ and sends a request. Since q_1 has sent "OK" to p_2, it does not send "OK" to p_3. Thus, p_3 cannot enter CS. However, p_3 could enter CS because p_1, which satisfies $gs(p_1) = gs(p_3)$, is currently entering CS. This is another type of unnecessary blocking.

In this scenario, p_1 and p_3 select their quorums, Q_1 and Q_3, such that $Q_1 \cap Q_3 = \phi$. Even if p_1 is currently entering CS, p_3 cannot know the fact from the processes it is contacting. By changing the definition of the quorum system, this type of unnecessary blocking can be avoided.

In the next section, we outline our algorithm which allows us to avoid unnecessary blocking.

4 New Algorithm for extended group mutual exclusion

4.1 Quorum system for extended group mutual exclusion

First, we define the condition to be satisfied to achieve unnecessary blocking free extended group mutual exclusion.

As shown in the example of first unnecessary blocking, a requesting process p must not decide its group before it sends a request. Thus, quorum system must be defined for each group set, not for each group, because p's group is not decided at making a request.

Let us define group set quorum system $\mathcal{C} = \{C_{G_1}, \ldots, C_{G_k}\}$, where C_{G_i} is a set of quorums for group set G_i. Requesting process p_i whose group set is G_i selects a quorum Q in C_{G_i} and sends a request to every member of Q.

We provide the conditions of the group set quorum system with the following theorem.

Theorem 1 *$\forall Q \in C_{G_i}$ and $\forall Q' \in C_{G_j} (1 \leq i, j \leq k)$, $Q \cap Q' \neq \phi$ must be satisfied to achieve extended group mutual exclusion without unnecessary blocking.* ■

(proof) First, suppose that $G_i \cap G_j = \phi$. In this case, requesting processes p_i and p_j, whose group sets are G_i and G_j, respectively, must not enter CS at the same time. If $Q \cap Q' = \phi$, p_i and p_j might receive "OK" from every member of Q and Q', respectively. Thus, $Q \cap Q' \neq \phi$ must be satisfied.

Next, suppose that $G_i \cap G_j \neq \phi$. Let g be a group in $G_i \cap G_j$. Now, suppose that there is no requesting process other than p_i, whose group set is G_i. Let $Q \in C_{G_i}$ be the quorum p_i selected. p_i can enter CS as a member of any group in G_i. Let us assume that p_i has set g as $gs(p_i)$. After that, p_j, whose group set is G_j, sends a request to every member of $Q' \in C_{G_j}$. Since $g \in G_j$, p_j must be able to enter CS. In order to achieve this, the information that p_i is currently entering CS as a member of g must arrive at p_j as a reply. Thus, Q and Q' must have at least one process in common. ■

The above theorem implies that the coterie for simple mutual exclusion can be used as the quorum set for any G_i. This fact makes the algorithm very simple, since we do not need to prepare a different quorum set for each G_i.

The coterie for simple mutual exclusion is defined as follows [4]. Coterie $C = \{Q_1, \ldots, Q_k\}$, where $Q_i \subseteq U$ and $Q_i \neq \phi$, satisfies the following two properties.
Intersection property: $\forall Q, Q' \in C, Q \cap Q' \neq \phi$.
Minimality property: $\forall Q, Q' \in C, Q \not\subseteq Q'$. ■

It is obvious that the coterie satisfies the condition in the theorem 1.

4.2 New group mutual exclusion algorithm

We begin by discussing the first type of unnecessary blocking. In order to avoid bad group selection, each requesting process must be able to set its group selection after it receives some replies from manager processes. We introduce a two-phase mutual exclusion algorithm. It was used in [10][19] to solve the generalized mutual exclusion problem. In the generalized mutual exclusion problem, there are multiple shared resources and each process may have different accessible resources. In the two-phase algorithm, each requesting process makes its decision after it receives "OK" from every process in a quorum. It then informs the processes in the quorum of its decision. With the generalized mutual exclusion, the decision is which resource it uses. With the extended group mutual exclusion, the decision is which group it selects as $gs(p)$.

First, the requesting process p sends "$Request$" to the processes in a quorum Q before it sets $gs(p)$. Each process q in Q, which received the request, replies "OK" to inform that p can enter CS or "$Enter(g)$" to inform that some pivot process p' which satisfies $g = gs(p') \in G(p)$ is currently entering CS. Using the replies, p enters CS if
(1) every process in Q replies "OK" or
(2) some process in Q replies "$Enter(g)$".

In case (1), p can set any group in $G(p)$ as $gs(p)$, since there is no other process that blocks p. p sends $gs(p)$ to the processes in Q. In case (2), p selects $gs(p) = g$ and enters CS. By rule (2), the first type of unnecessary blocking is avoided.

The outline of the procedure for the requesting process is as follows:

1. When p whose group set is G wants to enter CS, p selects a quorum $Q \in C$ and sends "$Request(G)$" to every process in Q.

2. There are two ways to enter CS.

   (a) When p receives "OK" from every process in Q, p arbitrarily selects one group $g \in G(p)$ as $gs(p)$, sends "$Lock(g)$" to every process in Q, and enters CS.

   (b) When p receives "$Enter(g)$" from some process in Q, p sets g as $gs(p)$ and enters CS.

3. When exiting CS, execute the following.

   (a) (entered CS by "OK") p sends "$Release$" to every process in Q.
   When p receives "$Finished$" from every process in Q, it sends "$Over$" to every process in Q.

   (b) (entered CS by "$Enter$") p sends "$NoNeed$" to the process "$Enter$" arrived. ■

The outline of the algorithm for the manager processes is shown later.

The exiting procedure when entered by "OK" is a little complicated. When "$Release$" arrives at a manager process, the process must not send "OK" to a waiting request immediately. Let us consider the following example. p_1, whose group set $G(p_1) = \{g_1\}$, uses $Q_1 = \{q_1, q_2\}$ and sends "$Request$". q_1 and q_2 send "OK" to p_1 and p_1 enters CS. After that, p_2 sends "$Request$" to $Q_2 = \{q_2, q_3\}$ and $G(p_2) = \{g_2\}(g_2 \neq g_1)$. q_3 sends "OK" to p_2. However, since q_2 has sent "OK" to p_1, p_2 receives no reply from q_2. Then, p_3, whose group set $G(p_3) = \{g_1\}$, sends "$Request$" to $Q_3 = \{q_1, q_3\}$. q_1 replies to p_3 with "$Enter(g_1)$", since p_1, whose group selection is g_1, is entering CS. Thus, p_3 can enter CS. Now, suppose that p_1 exits CS. p_1 sends "$Release$" to q_1 and q_2. If q_2 sends "OK" to p_2 immediately, p_2 enters CS, although p_3 is currently entering CS. Thus, extended group mutual exclusion is not achieved. Therefore, each process must not send "OK" to a waiting request until every process that entered CS by receiving "$Enter$" has exited.

A two-phase release procedure is used to achieve this. When "$Release$" arrives, each process stops sending "$Enter$" to further requests, waits for the exit of every process to which "$Enter$" was sent, and then replies "$Finished$". When the requesting process p receives "$Finished$" from every process in Q, it means all requests which entered CS by receiving "$Enter$" have exited. Then p sends "$Over$" to every member of Q. When "$Over$" arrives, each manager process sends "OK" to the highest priority waiting request.

The outline of the procedure for manager processes is as follows. Variable $status$ stores the current status of the process. $Status = vacant$ means there is no request, $waitlock$ means that it has sent "OK" to some process but "$Lock$" has not arrived, and $locked$ means "$Lock$" is received. Variable $group$ stores the current group when some process is entering CS.

In order to avoid starvation, each request has Lamport's logical clock [13]. A request with a smaller logical clock has a higher priority. Thus the oldest request will eventually have the highest priority and will be able to enter CS. The procedure to update the logical clock and assign it to each request is omitted in this procedure for simplicity.

The following is an outline of the procedure for manager processes.

1. When q receives $Request(G)$ from p, q inserts it in the queue Que.

   (a) If $status = vacant$, q sends "OK" to p and sets $status := waitlock$.

   (b) If $status = locked$ and $group \in G$, q sends "$Enter(group)$" to p.

2. When q receives "$Lock(g)$" from p, q sets $group := g$ and $status := locked$. q then sends "$Enter(g)$" to every waiting request in Que whose group set G satisfies $g \in G$.

3. When q receives "$Release$", q stops further sending of "$Enter$" (by changing $status$). And if there is no process to which "$Enter$" is sent, q replies "$Finished$".

4. When q receives "$NoNeed$" from p, q sends "$Finished$" to the process which sent "$Release$" to

q, if currently there is no process q has sent "$Enter$" or "OK".

5. When q receives "$Over$", q sets $status := vacant$ and tries to send "OK" to the highest priority request in Que. ■

In order to avoid deadlock, an additional mechanism is necessary. Assume that p_2's priority is higher than that of p_1. At q_1, "$Request$" arrives in the order p_1, p_2 and at q_2, it arrives in the order p_2, p_1. In this case, the "OK" sent from q_1 to p_1 must be canceled to avoid deadlock. The cancellation procedure is the same as that used for simple mutual exclusion in [16].

1. When process q receives "$Request(G)$" from p_2, if p has sent "OK" to p_1 but "$Lock$" has not arrived, and p_2's priority is higher than that of p_1, then q sends "$Cancel$" to p_1.
2. When p_1 receives "$Cancel$" from q, if it has not entered CS, it replies to q with "$Cancelled$" (and waits for the next arrival of "OK").
3. When q receives "$Cancelled$", q sends "OK" to the highest priority request in Que. ■

The meaning of the other variables used in Fig.1 are as follows. As regards each requesting process, $Rstatus$ stores the status of the request. $Rstatus = wait$ means it is waiting for "OK" or "$Enter$". In means that it is in the CS, out means that it has exited CS. The quorum currently in use is stored in Q. The set of processes from which "OK" has arrived (when making a request) or "$Finished$" has arrived (when releasing) is stored in K. Thus, if $K = Q$, the requesting process can enter CS (when making a request) or can send "$Over$" (when releasing).

Next, we describe the meaning of the variables for each manager process. Que is the priority queue of the requests ($Que[1]$ is the highest priority request). $Que[i]$ has entry $Que[i].pr$ (the requesting process) and $Que[i].G$ (the set of groups). The requesting process to which "OK" is sent is stored in variable $sentok$. Variable $using$ is the set of processes currently entering CS. The other values stored to variable $status$ are as follows: $waitcancel$ when "$Cancel$" is sent and $releasing$ when "$Release$" arrives.

Note that when p exits CS and makes another request, p might receive replies to the earlier request. p can ignore such old replies easily if the logical clock of each request is attached to every reply message. The procedure for ignoring such replies is omitted from Fig. 1 for simplicity.

4.3 Correctness of the algorithm

This subsection shows the correctness of the algorithm. First, it is shown that extended group mutual exclusion is achieved.

Theorem 2 *p_1 and p_2 never enter CS at the same time by the algorithm in Fig. 1 if $gs(p_1) \neq gs(p_2)$.* ■

(Proof) Suppose that the above situation occurs. Let $g_1 = gs(p_1)$, $g_2 = gs(p_2)$, and $Q_1(Q_2)$ be the quorum $p_1(p_2)$ uses. p_1 (and p_2) enters CS by (1) receiving "OK" from every member of Q_1 (Q_2) or (2) receiving "$Enter$" from some process in Q_1 (Q_2). In case (2), there is another process p'_1 (p'_2), whose group selection is $g_1(g_2)$, which enters CS before p_1 (p_2). p'_1 (p'_2) receives "OK" from every member of some quorum, say Q'_1 (Q'_2). Though p'_1 (p'_2) might have exited CS before p_1 (p_2) exits CS, the processes in Q'_1 (Q'_2) cannot send "OK" to any other process until p_1 (p_2) exits CS and sends "$NoNeed$".

In case (1), let $p'_1 = p_1$ ($p'_2 = p_2$) and $Q'_1 = Q_1$ ($Q'_2 = Q_2$).

$p'_1 \neq p'_2$ holds in any cases since $gs(p'_1) \neq gs(p'_2)$.

Now, every process in Q'_1 (and Q'_2) has sent "OK" to p'_1 (p'_2) at the same time. Since $Q'_1 \cap Q'_2 \neq \phi$ and each process sends "OK" to at most one process at the same time, this situation cannot occur. ■

Next, deadlock-freeness is briefly shown. Assume that a deadlock occurs. Assume that there is no new requesting process and every process that can enter CS enters and exits CS after the deadlock. In this situation, there is no process that waits for "$Over$" or "$NoNeed$", since every process that entered CS has exited CS. Thus, the deadlocked requesting processes send "$Request$" and wait for "OK". Therefore, this deadlock situation is just the same as that without the mechanism of entering CS by the "$Enter$" message, that is, the same one in the simple mutual exclusion algorithm in [16]. Since the algorithm in [16] is deadlock-free, the algorithm in Fig. 1 is also deadlock-free.

Lastly, starvation-freeness is shown.

Theorem 3 *No starvation occurs with the algorithm in Fig. 1* ■

(Proof) Assume that starvation occurs. Let p_1 be the highest priority request which cannot ever enter CS. Let Q_1 be the quorum p_1 selects. From the assumption, p_1 is the highest priority request that does not enter CS from some time t. Let t' be the time at which "$Request$" from p_1 arrives at every member of Q_1. Let us consider the system state after time $T = max(t, t')$. Each process $q \in Q_1$ must try to send "OK" to p_1 because p_1 has the highest priority. If q has not sent "OK" to any process, obviously it sends "OK" to p_1. If q has sent "OK" to another process, say p_2, q sends "$Cancel$" p_2. If p_2 has not entered CS, it replies "$Cancelled$" and thus, q will be able to send "OK" to p_1. If

p_2 has entered CS before the arrival of "$Cancel$", p_2 eventually exits CS. After time T, q does not send "$Enter$" to any other requests. In addition, every process that entered CS by receiving "$Enter$" from q (before T) also eventually exits CS. Thus, q eventually sends "$Finished$" to p_2 and thus, p_2 eventually sends "$Over$" to q. Therefore, q will be able to send "OK" to p_1 and no starvation occurs. ■

Lastly, we show unnecessary blocking freeness.

Theorem 4 *Unnecessary blocking never occurs with the algorithm in Fig. 1* ■

(Proof) Assume that unnecessary blocking occurs. Suppose that p_i, whose group selection is $gs(p_i)$, is currently in CS as a pivot. Suppose that p_i uses Q_i as its quorum. Since p_i is entering CS as a pivot, it receives "OK" from every member of Q_i. Thus, every manager process in Q_i satisfies that $status = locked$ and $group = gs(p_i)$.

Now, suppose that p_j, which satisfies $gs(p_i) \in G(p_j)$ makes a request using Q_j. Since $Q_i \cap Q_j \neq \phi$, there is at least one manager process, $q \in Q_i$, which receives this request. Since $status = locked$ and $group = gs(p_i)$, q sends "Enter($gs(p_i)$)" to p_j and p_j can enter CS. ■

4.4 Communication complexity

The communication complexity of the proposed algorithm is shown. Let $|Q|$ be the size of the smallest quorum in a coterie.

The no-exclusion case is that there is only one request at any time. This case is considered to the best case for the discussion of simple mutual exclusion.

(1) Process p sends "$Request$" to every member of Q.
(2) p receives "OK" from every member of Q.
(3) p sends "$Lock$" to every member of Q and enters CS.
(4) p exits CS and sends "$Release$" to every member of Q.
(5) p receives "$Finished$" from every member of Q.
(6) p sends "$Over$" to every member of Q. ■

In this case, the total number of messages is $6|Q|$.

Next, consider the best case. The best case is when a requesting process p_1, which satisfies $gs(p_1) = g_1$, is in CS for a very long time and there are many processes $p_2, p_3, \ldots$ during the period, whose group set satisfies $g_1 \in G(p_i) (i = 2, 3, \ldots)$. p_1's execution is the above no-execution case and its number of messages is $6|Q|$. The execution for $p_2, p_3, \ldots$ are as follows:

(1) p_i sends "$Request$" to every member of Q_i.
(2) p_i receives "$Enter(g_1)$" from $q_1 \in Q_i$, where $Q \cap Q_i = \{q_1\}$.
(3) Each process in $Q_i - \{q_1\}$ sends "OK" to p_i.
(4) p_i sends "$NoNeed$" to every member of $Q_i - \{q_1\}$ and enters CS.
(5) p_i exits CS and sends "$NoNeed$" to q_1. ■

The total number of messages for $p_i (i = 2, 3, \ldots)$ is $3|Q_i|$. Thus, the total number of messages per request is $3|Q| + \epsilon$, if $|Q_i| = |Q|$.

Lastly, consider the worst case, when the highest priority request arrives later.

(1) p sends "$Request$" to every member of Q.
(2) Each process $q_i \in Q$ has sent "OK" to another process p_i, whose priority is lower than that of p. q_i sends "$Cancel$" to p_i.
(3) p_i sends "$Cancelled$" to q_i.
(4) q_i receives "$Cancelled$" and sends "OK" to p.
(5) p receives "OK" from every member of Q. Thus, it sends "$Lock$" to every member of Q and enters CS.
(6) p exits CS and sends "$Release$" to every member of Q.
(7) p receives "$Finished$" from every member of Q.
(8) p sends "$Over$" to every member of Q.
(9) q_i receives "$Over$" and sends "OK" again to the process to which "$Cancel$" is sent (The messages for p_i to enter and exit CS are counted as the messages for p_i). ■

The total number of messages per request is $9|Q|$.

Although the number of messages in the worst case is larger than $2c + 1$ in [8], where c is the size of the quorum in the m-group quorum system, our algorithm avoids unnecessary blocking.

5 Conclusion

This paper has shown a new extended group mutual exclusion algorithm that prevents unnecessary blocking. One possible improvement of the proposed algorithm involves improving group selection. This paper's algorithm selects an arbitrary group when it can enter CS by receiving "OK". If the group is selected using the information on the waiting processes' group sets, the possibility for the waiting processes to enter CS increases. However, the worst case message complexity is unchanged by this modification.

Acknowledgement The authors would like to thank Mr. Yoshifumi Ooyama of NTT (currently NTT Advanced Technology) for his encouragement and suggestions.

References

[1] R. Atreya and N. Mittel, A Distributed Group Mutual Exclusion Algorithm using Surrogate-Quorums, Technical Report, The University of Texas at Dallas, 2003.

[2] S. Cantareli, A.K. Datta, F. Perit, V. Villain, Token Based group mutual exclusion for asynchronous rings, Proc. of 21st ICDCS, (2001), 691-694.

[3] S. Cantareli, A.K. Datta, F. Perit, V. Villain, Group Mutual Exclusion in Token Rings, Proc. of 8th Colloquium Struc-

tural Information and Communication Complexity, June 2001.

[4] H. Garcia-Molina, D. Barbara, How to assign votes in a distributed system, Journal of the ACM, 32, 4, (1985), 841-860.

[5] V. Hadzlilacos, A note on group mutual exclusion, Proc. of 20th PODC, (2001), 100-106.

[6] P. Jayanti, S. Petrovic, and K. Tan, Fair Group Mutual Exclusion, Proc. 22nd PODC, pp.275-284 (2003).

[7] Y.-J. Joung, Asynchronous group mutual exclusion, Distributed Computing, 13,4, (2000), 189-206.

[8] Y.-J. Joung, Quorum-based algorithm for group mutual exclusion, IEEE Trans. on Parallel and Distributed Systems, Vol.14, No.5, pp.463-476(May 2003).

[9] H. Kakugawa, S. Fujita, M. Yamashita, T. Ae, A distributed k-mutual exclusion algorithm using k-coterie, Information Processing Letters, 49, (1994), 213-238.

[10] H. Kakugawa, M. Yamashita, Local coteries and a distributed resource allocation algorithm, Trans. IPSJ, 37, 8 (1996), 145-159.

[11] M. Tonomura, S. Kamei, and H. Kakugawa, A Quorum-based Distributed Algorithm for Group Mutual Exclusion, Proc. 4th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, pp.74-74 (Aug. 2003).

[12] P. Keane, M. Moir, A simple local-spin group mutual exclusion algorithm, IEEE Trans. Parallel and Distributed Systems, 12, 7, (2001), 673-685.

[13] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of ACM, 21, 7, (1978), 558-565.

[14] Y. Manabe, R. Baldoni, M. Raynal, S. Aoyagi, k-Arbiter: A safe and general scheme for h-out of-k mutual exclusion, Theoretical Computer Science, 193, 1-2, (1998), 97-112.

[15] Y. Manabe and N. Tajima: (h,k)-Arbiters of h-out-of-k Mutual Exclusion Problem, Theoretical Computer Science, Vol.310, No.1-3 (2004).

[16] B.A. Sanders, The information structure of distributed mutual exclusion algorithms, ACM TOCS, 5, 3, (1987), 284-299.

[17] M. Singhal, A taxonomy of distributed mutual exclusion, Journal of Parallel and Distributed Computing, 18, 1, (1993), 94-101.

[18] R.K. Srimani, S.R. Das (eds.), Distributed mutual exclusion algorithms, IEEE Computer Society Press, 1992.

[19] S.-C. Sung, Y. Manabe, Coterie for generalized mutual exclusion problem, Trans. IEICE, E82-D, 5, (1999), 968-972.

[20] K. Vidyasankar, A highly concurrent group mutual l-exclusion algorithm, Proc. of 21th PODC, (2002), 130

[21] K.-P. Wu and Y.-J. Joung, Asynchorous Group Mutual Exclusion in Ring Networks, IEE Proc. Computers and Digital Techniques, Vol.147, No.1, pp.1-8 (2000).

```
program RequestingProcess(p:process);
var $Rstatus = wait$ : status of request ;
  $Q$ : set of process; /* quorum */
  $K$ : set of process; /* reply arrived */
  $G$ : set of group; /* current group set */

When $p$ (group set is $G$) wants to enter CS:
begin
  $Rstatus$ :=wait;
  Select arbitrary $Q$ from coterie;
  $K := \phi$;
  send "Request($G$)" to all $q \in Q$;
end; /* end of request initiation. */

At arrival of "OK" from $q$:
begin
  if $Rstatus = wait$ then begin
    $K := K \cup \{q\}$;
    if $K = Q$ then
      begin /* can enter CS */
        select arbitrary $g \in G$;
        send "Lock($g$)" to all $q \in Q$;
        $Rstatus := in$;
        .... /* in the CS */
        $Rstatus := out$;
        send "Release" to all $q \in Q$;
        $K := \phi$; /* waits for "Finished" */
      end/* end of $K = Q$ */
  end /* end of $Rstatus = wait$ */
end; /* end of "OK" arrival */

At arrival of "Enter($g$)" from $q$:
begin
  if $Rstatus = wait$ then begin
    send "NoNeed" to all $r \in Q - \{q\}$;
    $Rstatus := in$;
    .... /* in the CS */
    $Rstatus := out$;
    send "NoNeed" to $q$;
  end; /* end of $Rstatus = wait$ */
end; /* end of "Enter" arrival */

At arrival of "Cancel" from $q$:
begin
  if $Rstatus = wait$ then begin
    $K := K - \{q\}$;
    send "Cancelled" to $q$;
  end; /* end of $Rstatus = wait$ */
end; /* end of "Cancel" arrival */

At arrival of "Finished" from $q$:
begin
  $K := K \cup \{q\}$;
  if $K = Q$ then send "Over" to all $q \in Q$;
end; /* end of "Finished" arrival */
```

Figure 1(a). Algorithm for requesting process.

```
**program** ManagerProcess($q$:process);
**var** $status = vacant$ : status;
  $group$ : group; /* current group */
  $Que = null$ : priority queue of requests;
  $using = null$ : set of processes;
  $sentok = null$ : process; /* "OK" is sent. */

At arrival of "Request($G$)" from $p$:
**begin**
  insert the request to $Que$;
  /* assume that $Que[i]$ is the position. */
  $Que[i].pr := p$;
  $Que[i].G := G$;
  **if** $status = vacant$ **then begin**
    send "OK" to $p$;
    $sentok := p$;
    $status := waitlock$;
  **end**/* end of vacant */
  **else if** $status = locked$ **then begin**
    **if** $group \in G$ **then begin**
      send "Enter($group$)" to $p$;
      $using := using \cup \{p\}$;
    **end**
  **end**/* end of locked */
  **else if** $status = waitlock$ **then begin**
    **if** $i = 1$ **then begin**
      /* $i = 1$ : $i$'s priority is the highest */
      send "Cancel" to $sentok$;
      $status := waitcancel$;
    **end**
  **end** /* end of waitlock */
**end**; /* End of "Request" arrival */

At arrival of "Lock($g$)" from $p$ ($p = Que[i].pr$):
**begin**
  $using := \{p\}$;
  $status := locked$;
  $group := g$;
  **for** every request $Que[k] (k \neq i)$ such that
    $group \in Que[k].G$ **do begin**
      send "Enter($group$)" to $Que[k].pr$;
      $using := using \cup \{Que[k].pr\}$;
    **end**; /* end of do */
**end**; /* end of "Lock" arrival */

At arrival of "Release" from $p$ ($p = Que[i].pr$):
**begin**
  $status := releasing$;
  remove entry $Que[i]$;
  $using := using - \{p\}$;
  **if** $using = \phi$ **then** send "Finished" to $p$;
**end**; /* end of "Release" arrival */

At arrival of "NoNeed" from $p$ ($p = Que[i].pr$):
**begin**
  remove entry $Que[i]$;
  **if** $p = sentok$ **then** SendOK
    **else if** $p \in using$ **then begin**
      $using := using - \{p\}$;
      **if** $using = \phi$ **then** send "Finished" to $sentok$;
    **end**
**end**; /* end of "NoNeed" arrival */

At arrival of "Cancelled" from $p$:
**begin**
  SendOK;
**end**;

At arrival of "Over" from $p$:
**begin**
  SendOK;
**end**;

**procedure** SendOK; /* "OK" is released. */
**begin**
  **if** $Que$ is not empty **then begin**
      /* $Que[1]$ is the highest priority request */
    Send "OK" to $Que[1].pr$;
    $sentok := Que[1].pr$;
    $status := waitlock$;
  **end** /* end of $Que$ is not empty */
  **else** $status := vacant$;
**end**; /* end of procedure SendOK */
```

Figure 1(b). Algorithm for manager process.