(h,k)-arbiters for h-out of-k Mutual Exclusion Problem

Yoshifumi Manabe
NTT Science and Core Technology Laboratory Group
manabe@theory.brl.ntt.co.jp

Naka Tajima[*]
University of Aizu
m5011214@u-aizu.ac.jp

Abstract

h-out of-k mutual exclusion is a generalization of 1-mutual exclusion problem, where there are k units of shared resources and each process requests $h(1 \leq h \leq k)$ units at the same time. Though k-arbiter has been shown to be a quorum-based solution to this problem, quorums in k-arbiter are much larger than these in the 1-coterie for 1-mutual exclusion. Thus, the algorithm based on k-arbiter needs many messages. This paper defines two (h,k)-arbiters for h-out of-k mutual exclusion: a uniform (h,k)-arbiter and a $(k+1)$-cube (h,k)-arbiter. The quorums in each (h,k)-arbiter are not larger than the ones in the corresponding k-arbiter; consequently using the (h,k)-arbiters is more efficient than using the k-arbiters. Uniform (h,k)-arbiter is an optimal generalization of the majority coterie for 1-mutual exclusion. $(k+1)$-cube (h,k)-arbiter is a quasi-optimal generalization of square grid coterie for 1-mutual exclusion.

1. Introduction

Mutual exclusion is a fundamental problem in distributed systems. When resources, such as files, printers, and communication lines, are shared by multiple processes, they must be allocated so that each resource is not allocated to multiple processes at the same time. Many algorithms for distributed systems have been proposed to solve this mutual exclusion problem [19][20]. Firstly considered was the case where there is one shared resource, i.e., a *1-mutual exclusion problem*. More recently considered was the case where there are k units of an identical shared resource [16], i.e., a *k-mutual exclusion problem*. Many algorithms for the k-mutual exclusion problem have been reported [1][2][3][5][6][7][8][9][12][14][15][16][21][22].

In the k-mutual exclusion problem, each process can request one unit of the shared resource at the same time. The h-out of-k mutual exclusion approach allows every process to request h $(1 \leq h \leq k)$ units of the shared resource at the same time [17]. One example of this type of mutual exclusion is the sharing of a communication bandwidth. A communication line with a fixed bandwidth k is shared by several processes. Each process does audio and video communication over the line. Because the bandwidths necessary for audio and video communication differ greatly, the bandwidth needed differs from request to request.

While the h-out of-k mutual exclusion problem can be solved by using a k-mutual exclusion algorithm and executing h requests when a process needs h units of the shared resource, two problems arise. The first problem is poor efficiency. It would be better for the requesting process to make only one request rather than making multiple requests. The second problem is potential deadlock. Assume that there are k units and two processes, p_1 and p_2. Process p_1 requests h_1 units and process p_2 requests h_2 units, but $h_1 + h_2 > k$. If process p_1 can only obtain $h'_1(< h_1)$ units, process p_2 can only obtain $h'_2(< h_2)$ units, and $h'_1 + h'_2 = k$, a deadlock is created. An additional mechanism is thus needed to avoid this deadlock in order to solve the h-out of-k mutual exclusion when using a k-mutual exclusion algorithm. Thus, this paper discusses obtaining h units by making one request.

Distributed mutual exclusion algorithms can be classified into two main groups: token-based algorithms and quorum-based algorithms. In the former, a *permission* is represented by a token. If a process has a token, it can access the shared resource. In the latter, a permission is divided into pieces that are distributed among the processes. If a process can obtain enough pieces from the other processes to construct a permission, it can access the shared resource. A set of processes whose pieces can construct a permission is called a *quorum*. Although token-based algorithms generally show good performance regarding the number of messages exchanged to obtain the shared resource, they suffer from poor failure resiliency. In contrast, quorum-based algorithms tolerate failures of nodes and network partitions. We thus focused on using quorum-based algorithms for the h-out of-k mutual exclusion.

The k-arbiter has been shown to be a quorum-based distributed algorithm for the h-out of-k mutual exclusion [13].

[*]Part of this work was done while the second author was staying at NTT laboratories.

However, each quorum is larger than that in a 1-coterie for the 1-mutual exclusion algorithm. We thus looked at reducing the size of the quorums. The quorum-based distributed algorithm based on k-arbiter uses the same quorum for any request. We propose a new quorum based solution that uses an (h, k)-arbiter. This (h, k)-arbiter is a quorum set for each h $(1 \leq h \leq k)$, $\{Q_{h,k} \mid 1 \leq h \leq k\}$, where $Q_{h,k}$ is a set of quorums. A process uses a quorum in $Q_{h,k}$ when it wants to use h units of the shared resource.

We developed two such arbiters. One is a uniform (h, k)-arbiter, which is a generalization of the uniform k-arbiter. The other one is a $(k+1)$-cube (h, k)-arbiter, which is a generalization of the $(k+1)$-cube k-arbiter. The quorums in each (h, k)-arbiter are not larger than these in the corresponding k-arbiter. Thus, h-out of-k mutual exclusion can be solved more efficiently by using an (h, k)-arbiter than by using a k-arbiter. Moreover, each quorum in $Q_{k,k}$ of the (h, k)-arbiter can be no smaller than that in the 1-coterie for 1-mutual exclusion because obtaining all units is exactly the same as obtaining the shared resource in 1-mutual exclusion. Each quorum in $Q_{k,k}$ of the uniform (h, k)-arbiter is exactly the same size as that in the majority coterie for 1-mutual exclusion. Thus, the uniform (h, k)-arbiter can be considered to be an optimal generalization of the majority coterie. Each quorum in the $(k+1)$-cube (k, k)-arbiter is about the same size as that in the square grid coterie [11] for 1-mutual exclusion, whose size is minimum. Thus, $(k+1)$-cube (k, k)-arbiter can be considered to be a quasi-optimal generalization of the square grid coterie.

2. Problem definition and previous results

2.1. Distributed system

A distributed system is a set $U = \{p_1, p_2, \ldots, p_n\}$ of n processes. These processes communicate by exchanging messages. Each pair of processes is connected by a logical channel, and the message delay is unpredictable but finite. Moreover, each channel is assumed to have infinite capacity, to be error-free, and FIFO (First-In, First-Out). Processes do not share either a common clock or a shared memory. No bound exists to the relative speed of processes. The processes fail in accordance with the fail-stop model [18], and a failure can be detected by the other processes.

2.2. h-out of-k mutual exclusion problem

The h-out of-k mutual exclusion problem is defined as follows [17]. There are k identical units of a resource that is shared by the processes in U. Each unit must not be allocated to more than one process at the same time. A process requests h $(1 \leq h \leq k)$ units of the resource all at once and, to avoid deadlock, the process is blocked until it gets all of its requested units. The process can then start using the units; when finished, it releases them all at once. If $h = 1$ for every request, the h-out of-k mutual exclusion problem corresponds to the k-mutual exclusion one; moreover, if $k = 1$ we get 1-mutual exclusion.

The h-out of-k mutual exclusion algorithm must satisfy two properties:

safety : each unit of the resource may be used by at most one process at any given time;

liveness : all requests must be eventually satisfied.

2.3. Quorums, 1-coteries, and k-arbiters

The concept of coteries was introduced [4] to achieve safety for the 1-mutual exclusion problem:

Definition 1 *A quorum q under U is a non-empty subset of U.*

A set of quorums $Q = \{q_1, q_2, \ldots, q_m\}$ is a 1-coterie under U iff the following properties hold:

- *Intersection: For any pair of quorums, $q_i, q_j \in Q$ :: $q_i \bigcap q_j \neq \emptyset$;*
- *Minimality: For any pair of distinct quorums, $q_i, q_j \in Q$ :: $q_i \not\subseteq q_j$.* ■

An outline of the mutual exclusion algorithm using 1-coteries is as follows. A process selects a quorum from a 1-coterie, sends a request to each process in the quorum, and waits to receive permissions from them. After receiving the permissions, the process can use the shared resource. Safety is guaranteed from the intersection property. Minimality is introduced to remove unnecessary elements. If $q_i \subseteq q_j$, q_j is unnecessary because using q_j needs more communication than using q_i.

We briefly mention two examples of 1-coteries that will be used in the following.

1. Majority coteries: $Q = \{q \subset U \mid |q| = \lfloor n/2 \rfloor + 1\}$, where $n = |U|$.
2. Square grid coteries [11]: Assume that the processes in U are structured into a square grid $(1 \ldots \sqrt{n}, 1 \ldots \sqrt{n})$.
   $Q = \{q_{i,j} \mid 1 \leq i \leq \sqrt{n}, 1 \leq j \leq \sqrt{n}\}$, where
   $q_{i,j} = \{(x, y) \mid x = i\} \cup \{(x, y) \mid y = j\}$.
   The size of each quorum is $O(\sqrt{n})$.

In order to solve k-mutual exclusion, a k-arbiter has been defined [13]. Its intersection property differs from that for 1-mutual exclusion as follows.

Definition 2 *A set of quorums $Q = \{q_1, q_2, \ldots, q_m\}$ is a k-arbiter under U iff the following properties hold:*

- *Intersection: For any $(k+1)$ quorums, $q_{i_1}, q_{i_2}, \ldots, q_{i_{k+1}} \in Q$ ::* $\bigcap_{1 \le j \le k+1} q_{i_j} \neq \emptyset$*;*
- *Minimality: For any pair of distinct quorums, $q_i, q_j \in Q$ :: $q_i \not\subseteq q_j$.* ■

When $k = 1$, the k-arbiter becomes a 1-coterie. Two examples of k-arbiters are as follows [13].

1. Uniform k-arbiter: $Q = \{q \subset U \mid |q| = \lfloor k \cdot n/(k+1) \rfloor + 1\}$.
2. $(k+1)$-cube k-arbiter: Suppose that $n = a^{k+1}$ for some integer a. The processes are structured into a $(k+1)$-dimensional hypercube. Each process is represented by $(x_1, x_2, \ldots, x_{k+1})(0 \le x_i \le a-1, 1 \le i \le k+1)$.

   $Q = \{q^{b_1,\ldots,b_{k+1}} \mid 0 \le b_j \le a-1, 1 \le j \le k+1\}$, where $q^{b_1,\ldots,b_{k+1}} = \{(x_1, x_2, \ldots, x_{k+1}) \mid x_1 = b_1\}$ $\cup\{(x_1, x_2, \ldots, x_{k+1}) \mid x_2 = b_2\} \cup \ldots$ $\cup\{(x_1, x_2, \ldots, x_k, x_{k+1}) \mid x_k = b_k\}$ $\cup\{(x_1, x_2, \ldots, x_{k+1}) \mid x_{k+1} = b_{k+1}\}$.

   The size of each quorum is at most $(k+1)n^{\frac{k}{k+1}}$.

When $n \neq a^{k+1}$, the above construction is easily modified to obtain a k-arbiter [13].

The uniform k-arbiter becomes the majority coterie when $k = 1$. The $(k+1)$-cube k-arbiter becomes the square grid coterie when $k = 1$. Note that the lower bound on the size of each quorum of the symmetric k-arbiter is $O(n^{\frac{k}{k+1}})$ [13]. Thus, a $(k+1)$-cube k-arbiter satisfies the lower bound when k is a constant.

An outline of the k-mutual exclusion algorithm using a k-arbiter is as follows. When a process wants to use h units of the shared resource, it selects a quorum from a k-arbiter, sends a request to each process in the quorum, and waits to receive permissions from them. After receiving the permissions, the process can use the shared resource. When a process receives a request to use h units from a process p, it sends permission to p if the number of units in the requests it sends permission is not greater than $k - h$. Safety is guaranteed from the intersection property. Minimality is exactly the same as that for the 1-coterie.

3. (h, k)-arbiter

3.1. Definition of (h, k)-arbiter

We assume that each process uses a different arbiter for each h $(1 \le h \le k)$. An (h, k)-arbiter, $\mathcal{Q}_{h,k}$, is a set of arbiters $\{Q_{h,k} \mid 1 \le h \le k\}$. If a process p wants to use h units of the shared resource, p selects a quorum $q_{h,k} \in Q_{h,k}$ and executes the same procedure as that for using a k-arbiter. The algorithm to send a permission is exactly the same as that for a k-arbiter. The intersection property of the (h, k)-arbiter differs from that for a k-arbiter, as shown in the following.

A *set* has no repeated elements, while a *bag* may have repeated elements: $\{1, 2, 2, 3\}$ is a bag of four elements.

Definition 3 *A request pattern of h-out of-k mutual exclusion, r_k, is a bag of positive integers up to k.* ■

The h-out of-k mutual exclusion is omitted if it is obvious.

Definition 4 *A request pattern r_k is conflicting iff $\sum_{h \in r_k} h \ge k+1$. A conflicting request pattern r_k is critical iff $\forall h \in r_k$, $r_k - \{h\}$ is not conflicting.* ■

For example, $r_4^1 = \{2, 2, 3\}$ and $r_4^2 = \{1, 1, 1, 2\}$ are conflicting request patterns for $k = 4$; r_4^1 is not critical because $\{2, 3\}$ is also a conflicting request pattern, while r_4^2 is critical.

Definition 5 *For a request pattern r_k, a bag $\{\exists q \in Q_{h,k} \mid h \in r_k\}$ is called a quorum assignment for r_k. Let $QA(r_k)$ be the set of all quorum assignments for r_k.* ■

A quorum assignment means the case when each process selects a quorum in $Q_{h,k}$ and sends a request. Because $|Q_{h,k}| \ge 1$, there can be multiple quorum assignments for a given request pattern.

Theorem 1 *(Intersection Property) The safety property is guaranteed iff*

$$\forall qa \in QA(cr_k), \bigcap_{q \in qa} q \neq \emptyset \qquad (1)$$

is satisfied for any critical conflicting request pattern cr_k. ■

Note that the minimality property is just the same as the one for the 1-coterie.

(Proof) First, assume that $\exists qa \in QA(cr_k)$, $\bigcap_{q \in qa} q = \emptyset$ is satisfied for a conflicting request pattern cr_k. Since $\bigcap_{q \in qa} q = \emptyset$, no process receives every request in cr_k. Since cr_k is critical, for any $h \in cr_k$, $cr_k - \{h\}$ is not conflicting. Thus, since $\sum_{h' \in cr_k - \{h\}} h' \le k$, every process sends a permission to every request. Therefore, every request in cr_k receives enough permissions to use the shared resource at the same time. Therefore, safety is not guaranteed.

Next, assume that $\forall qa \in QA(cr_k)$, $\bigcap_{q \in qa} q \neq \emptyset$ is satisfied for any critical conflicting request pattern cr_k. We show that the processes cannot use the shared resource at the same time for any conflicting request pattern r_k. For any conflicting request pattern r_k, there is a critical conflicting request pattern cr_k satisfying $cr_k \subseteq r_k$. Such a cr_k can be obtained as follows. If r_k is critical, set $cr_k := r_k$ and terminate the procedure. If not, there is $h \in r_k$ satisfying $r_k - \{h\}$ is conflicting. Remove h from r_k. New r_k is also a conflicting request pattern and go back to the top of the procedure.

This procedure terminates for every r_k since r_k is a finite bag. Thus, there is a critical conflicting request pattern cr_k satisfying $cr_k \subseteq r_k$. From the assumption, there is a process p that receives every request in cr_k. Since p can send permissions for up to k requested units at the same time, p does not send a permission to every request in cr_k. Thus, all requests in cr_k cannot use the shared resource at the same time, meaning that the requests in $r_k (\supseteq cr_k)$ do not use the shared resource at the same time, which guarantees the safety property. ■

Note that the condition in Theorem 1 becomes the intersection property for k-coterie when $cr_k = \{1, 1, \ldots, 1\}$. Thus, the intersection property for (h, k)-arbiter includes the one for k-arbiter.

The following property is satisfied for critical conflicting request patterns.

Theorem 2 *Critical conflicting request pattern cr_k satisfies $\sum_{h \in cr_k} h \leq k + \alpha$, where $\alpha = \min_{h \in cr_k} h$.* ■

(Proof) If a critical conflicting requesting pattern cr_k satisfies $\sum_{h \in cr_k} h \geq k + 1 + \alpha$, $cr_k - \{\alpha\}$ satisfies $\sum_{h \in cr_k - \{\alpha\}} h \geq k + 1$ and $cr_k - \{\alpha\}$ is conflicting. This implies that cr_k is not critical. ■

3.2. Uniform (h, k)-arbiter

The uniform (h, k)-arbiter is defined as follows.

$$Q_{h,k} = \{q \subset U \mid |q| = \left\lfloor \frac{k \cdot n}{k + h} \right\rfloor + 1\} \ .$$

Theorem 3 *The uniform (h, k)-arbiter satisfies the conditions of the (h, k)-arbiter.*

(Proof) In order to prove that the condition of Theorem 1 is satisfied, we show that there is a process that receives every request. If a process requesting h units selects $q_h \in Q_{h,k}$, $n - |q_h|$ processes do not receive the request from the process. Thus, we must show that $\forall qa \in QA(cr_k), \sum_{q \in qa}(n - |q|) < n$ for any critical conflicting request pattern cr_k. From the definition of $Q_{h,k}$, this inequality can be written as

$$\sum_{h \in cr_k} (n - \left\lfloor \frac{k \cdot n}{k + h} \right\rfloor - 1) < n \ .$$

Since $\left\lfloor \frac{k \cdot n}{k+h} \right\rfloor \geq \frac{k \cdot n}{k+h} - \frac{k+h-1}{k+h}$,

$$\sum_{h \in cr_k} (n - \left\lfloor \frac{k \cdot n}{k + h} \right\rfloor - 1) \leq \sum_{h \in cr_k} \frac{h \cdot n - 1}{k + h} \ .$$

When $\min_{h \in cr_k} h = \alpha$, $\sum_{h \in cr_k} h \leq k + \alpha$ (from Theorem 2). Thus,

$$\sum_{h \in cr_k} \frac{h \cdot n - 1}{k + h} \leq \sum_{h \in cr_k} \frac{h \cdot n - 1}{k + \alpha}$$
$$\leq \frac{(k + \alpha)n - |cr_k|}{k + \alpha} < n \ .$$

Therefore, Eq. (1) holds. Minimality holds since $|q| = |q'|$ for any two $q, q' \in Q_{h,k}$. ■

The quorums in the uniform (h, k)-arbiter are no larger than these in the uniform k-arbiter. Thus, the uniform (h, k)-arbiter is better than the uniform k-arbiter.

Assume that the number of units h in every request satisfies $h = k$. In this case, h-out of-k mutual exclusion becomes 1-mutual exclusion, meaning that every process tries to access the unique imaginary resource consisting of k units. Thus, the quorums in $Q_{k,k}$ cannot be smaller than these in a 1-coterie. Since the uniform (h, k)-arbiter is defined so that the requesting process can use the shared resource if the number of permissions exceeds a threshold, i.e., defined exactly the same as the majority coterie for 1-mutual exclusion, the quorums in $Q_{k,k}$ cannot be smaller than these in the majority coterie. The size of the quorums in $Q_{k,k}$ is $\lfloor \frac{n}{2} \rfloor + 1$, which is exactly the same as that of those in the majority coterie. Thus the uniform (h, k)-arbiter is an optimal generalization of the majority coterie for 1-mutual exclusion.

3.3. $(k + 1)$-cube (h, k)-arbiter

The $(k + 1)$-cube (h, k)-arbiter is defined as follows. First, assume that $n = a^{k+1}$ for some integer a. Each process corresponds to a node on a $(k + 1)$-dimensional hypercube, which can be represented by $\{(x_1, x_2, \ldots, x_{k+1}) \mid 0 \leq x_i \leq a - 1, 1 \leq i \leq k + 1\}$.

Let

$$z_h = \left\lfloor \frac{1 + k}{1 + \frac{k}{h}} \right\rfloor \ .$$

$$Q_{h,k} = \{q_{h,k}^{b_1, \ldots, b_{k+1}} \mid 0 \leq b_i \leq a - 1, 1 \leq i \leq k + 1\} \ ,$$

where

$$q_{h,k}^{b_1, \ldots, b_{k+1}} = \bigcup_{0 \leq j \leq k+1-z_h} \{(x_1, x_2, \ldots, x_{k+1}) \mid$$
$$x_{i+j} = b_{i+j}, 1 \leq i \leq z_h\} \ .$$

$q_{h,k}^{b_1,\ldots,b_{k+1}}$ consists of $(k+2-z_h)$ subcubes whose dimensions are $(k+1-z_h)$. Therefore, the size of each quorum in $Q_{h,k}$, $|q_{h,k}^{b_1,\ldots,b_{k+1}}|$, is no larger than $(k+2-z_h)a^{k+1-z_h}$.

Theorem 4 *The $(k+1)$-cube (h,k)-arbiter satisfies the conditions of the (h,k)-arbiter.* ■

(Proof) In order to prove that the condition of Theorem 1 is satisfied, we show that there is a process that receives every request. Let $\{q_{h_1,k}^{b_1^1,b_2^1,\ldots,b_{k+1}^1}, q_{h_2,k}^{b_1^2,b_2^2,\ldots,b_{k+1}^2}, \ldots, q_{h_\ell,k}^{b_1^\ell,b_2^\ell,\ldots,b_{k+1}^\ell}\}$ be a quorum assignment for a critical conflicting request pattern. From Theorem 2, $\sum_{i=1}^{\ell} h_i \leq k+\alpha$, where $\alpha = \min_{1\leq i\leq \ell} h_i$. Let $y_0 = 0$ and $y_i = \sum_{j=1}^{i} z_{h_j} (1 \leq i \leq \ell)$. From the definition of z_h, $y_\ell = \sum_{j=1}^{\ell} \left\lfloor \frac{1+k}{1+\frac{k}{h_j}} \right\rfloor \leq \sum_{j=1}^{\ell} \frac{1+k}{1+\frac{k}{h_j}} = (1+k)\sum_{j=1}^{\ell} \frac{h_j}{h_j+k} \leq (1+k)\sum_{j=1}^{\ell} \frac{h_j}{\alpha+k} \leq (1+k)\frac{k+\alpha}{\alpha+k} = (1+k)$.

Now consider a process $v = (b_1^1, b_2^1, \ldots, b_{y_1}^1, b_{y_1+1}^2, b_{y_1+2}^2, \ldots, b_{y_2}^2, \ldots, b_{y_{i-1}+1}^i, b_{y_{i-1}+2}^i, \ldots, b_{y_i}^i, \ldots, b_{y_{\ell-1}+1}^\ell, b_{y_{\ell-1}+2}^\ell, \ldots, b_{y_\ell}^\ell, c_{y_\ell+1}, c_{y_\ell+2}, \ldots, c_{k+1})$, where $c_j (y_\ell + 1 \leq j \leq k+1)$ is an arbitrary number. If $y_\ell = k+1$, c_j does not exist.

$v \in q_{h_i,k}^{b_1^i,b_2^i,\ldots,b_{k+1}^i}$ is satisfied for every i $(1 \leq i \leq \ell)$, since subcube $\{(x_1, x_2, \ldots, x_{k+1}) \mid x_{y_{i-1}+j} = b_{y_{i-1}+j}^i, 1 \leq j \leq z_h\}$ is contained in $q_{h_i,k}^{b_1^i,b_2^i,\ldots,b_{k+1}^i}$.

Therefore, Eq. (1) holds. Minimality holds since $|q| = |q'|$ for any two $q, q' \in Q_{h,k}$. ■

If $n \neq a^{k+1}$, we can modify the above quorum definition as follows. Assume that $(a-1)^{k+1} < n < a^{k+1}$. The $(k+1)$-tuple $(b_1, b_2, \ldots, b_i, \ldots, b_{k+1})$ $(0 \leq b_i \leq a-1, 1 \leq i \leq k+1)$ represents process $\sum_{i=1}^{k+1} b_i \cdot a^{i-1} (\text{mod } n)$. Construct the $(k+1)$-cube (h,k)-arbiter for these a^{k+1} tuples. Based on this relation between $(k+1)$-tuples and processes, it is obvious that the above quorum definition satisfies the intersection property. Minimality is satisfied by removing q' if $q \subset q'$ for some $q, q' \in Q_{h,k}$.

The quorums in the $(k+1)$-cube (h,k)-arbiter are not larger than those in the $(k+1)$-cube k-arbiter. Thus, the $(k+1)$-cube (h,k)-arbiter is better than the $(k+1)$-cube k-arbiter. The size of the quorums in $Q_{k,k}$ is $(\lceil \frac{k+1}{2} \rceil + 1)a^{\lceil \frac{k+1}{2} \rceil}$, which is close to the minimum value $O(\sqrt{n})$.

4. Distributed mutual exclusion algorithm

4.1. Algorithm description

In this section we present a distributed algorithm for h-out of-k mutual exclusion that uses an (h,k)-arbiter. The intersection property guarantees safety. The distributed algorithm used to achieve liveness is the same as that for the k-arbiter. Its outline is as follows.

Each process maintains a Lamport's logical clock [10]. Let the value of process p's current clock be c_p. When p sends message m, the current value of c_p is piggybacked on m. Let c_m be the value of the clock piggybacked on m when p receives message m. p updates $c_p := max(c_p, c_m) + 1$.

Each request is a tuple (h, p, c), where h is the number of units, p is the requesting process, and c is the value of the logical clock when p initiated the request. A priority is assigned to every request. The priority of request (h, p, c) is higher than that of (h', p', c') if $c < c'$ or ($c = c'$ and $p < p'$). A total order is thus given to the requests.

When p initiates request (h, p, c), it selects a quorum $q \in Q_{h,k}$ and sends a "request (h, p, c)" message to every process in q. If p receives an "OK" response from every process in q, it can use h units of the shared resource. When p finishes using the units, it sends a "release" message to every process in q. Note that a "cancel" message might be received during waiting for "OK"s. The procedure for this is shown below.

Each process has initially k permissions. Let x_p be the permissions p currently has. Each process also maintains a priority queue of requests. It tracks the status of each request: "wait", "OK", or "cancel". The priority of requests is defined as described above. Initially, the queue is empty. When p receives "request (h', p', c')", it inserts the request in its priority queue and sets the status to "wait".

If the request (h', p', c') satisfies that the total units requested in the higher priority requests in the queue is no more than $k - h'$, and $x_p \geq h'$, p sends an "OK" message to p', changes the request's status to "OK", and executes $x_p := x_p - h'$.

There may be a request (h'', p'', c'') in the priority queue whose status is "OK" and the total units requested in higher priority requests is now more than $k - h''$ as a result of inserting new requests in the priority queue. Note that multiple requests might satisfy this condition as a result of inserting one request. The "OK" to these requests must be cancelled, otherwise a deadlock might occur. Thus, p sends a "cancel" message to p'' to cancel the "OK" and changes the status of the request to "cancel".

When p'' receives the "cancel" message from p, it sends a "cancelled" message back to p if it has not yet received "OK" from every process in q. It then waits for another "OK" message from p.

When p receives a "cancelled" message from p'', it changes the status of request (h'', p'', c'') to "wait". p executes $x_p := x_p + h''$ and tries to send an "OK" message to the higher priority requests based on the above condition for sending an "OK" message.

When p receives a "release" message from p'', it removes the request (h'', p'', c'') from its priority queue, executes $x_p := x_p + h''$, and tries to send an "OK" message to the other requests based on the above condition for sending an "OK" message. The detail of the algorithm is shown in Fig. 1.

4.2. Correctness of algorithm

The proof of the correctness of this algorithm is shown.

Theorem 5 *No deadlock occurs in the h-out of-k mutual exclusion algorithm.* ■

(Proof) Let us consider the wait-for graph G. Each node in G represents a process. A directed edge (p, p') exists in G iff both of the following conditions hold for a process r.

- p has sent "request" to r and is waiting for "OK" from r.
- r has sent "OK" to p' and it is not canceled by "cancel".

First we show that when a deadlock occurs, there is a permanent directed cycle in G. Without loss of generality, we assume that after a deadlock occurs, all processes that can enter the critical section have existed from the critical section and no new request occurs. In this case, any node without an edge in G is not involved in the deadlock. Thus G has at least one edge. If there is no directed cycle in G, then there is a process p with an incoming edge and no outgoing edge. Since p has no outgoing edge, every process which receives "request" from p has not sent "OK" to any other process. Thus p receives "OK" and p can enter the critical section. This contradicts that a deadlock occurs.

Next we show that there is no permanent directed cycle in wait-for graph G during the execution of the algorithm. Let us assume that there is a permanent directed cycle $p_0, p_1, \ldots, p_{m-1}, p_0$ in G, that is, there are directed edges $(p_i, p_{i+1(mod\ m)})(0 \leq i \leq m-1)$. Let $r_i(0 \leq i \leq m-1)$ be the process that has sent "OK" to $p_{i+1(mod\ m)}$ and received "request" from p_i.

The sum of the units of the requests whose priority is higher than or equal to that of p_i is more than k at r_i. Otherwise, r_i would have sent "cancel" to lower priority requests, waited until enough tokens were available, and sent "OK" to p_i. Since the "OK" to $p_{i+1(mod\ m)}$ is not canceled by r_i, the sum of the units of the requests whose priority is higher than or equal to that of $p_{i+1(mod\ m)}$ is no more than k at r_i. Thus, the priority of p_i is lower than that of $p_{i+1(mod\ m)}$. Since there is a directed cycle, p_0's priority would have to be lower than p_0's priority. This contradicts the definition of the priority. Therefore, there is no permanent directed cycle in G and no deadlock occurs. ■

Theorem 6 *No starvation occurs in the h-out of-k mutual exclusion algorithm.* ■

(Proof) Assume that there is a requesting process p that cannot enter the critical section forever. The request has a pair (p, c_p) as its priority. Let q be the (h, k)-arbiter selected by p. Since p cannot enter the critical section forever, at least one process $r \in q$ receives an infinite number of requests whose priority is higher than p, after receipt of p's request. Note that after receiving the request from p, the value of r's logical clock c_r satisfies $c_r > c_p$. Thus, when some requesting process u receives "OK" from r, u's logical clock value is greater than c_p and u cannot send any more request whose priority is higher than (p, c_p). Thus, there cannot be an infinite number of requests with a priority higher than (p, c_p). Therefore, p can enter the critical section. ■

4.3. Message complexity

Let us enumerate the number of messages sent per request. The best case is that there is no conflict.

1. p sends "request" to every process in some $q \in Q_{h,k}$.
2. Every process in q sends "OK" to p.
3. p enters the critical section and exits. p sends "release" to every process in q.

Thus, the message complexity is $3|q_{h,k}|$ per request, where $|q_{h,k}|$ is the size of the largest quorum in $Q_{h,k}$.

Next consider the worst case. The worst case is that there is conflict and lower priority requests must be canceled.

1. p sends "request" to every process in some $q \in Q_{h,k}$.
2. Each process $u \in q$ has sent k "OK"s to other requests whose requesting units are 1. The request from p has a priority higher than these requests. Thus, u sends "cancel" to h requesting processes to cancel the "OK"s.
3. Each process replies "cancelled" to u. Thus, u sends "OK" to p.
4. p enters the critical section and exits. p sends "release" to every process in q.
5. u receives "release" and sends "OK" again to the canceled processes.

Note that after p receives "OK", it may be canceled by another process r's request. Messages for this procedure are counted in the procedure for r. Thus, the message complexity of this case is $(3h + 3)|q_{h,k}|$ per request.

5. Concluding remarks

We have defined two (h, k)-arbiters for h-out of-k mutual exclusion: a uniform (h, k)-arbiter and a $(k+1)$-cube (h, k)-arbiter. The quorums in each (h, k)-arbiter are not larger than the ones in the corresponding k-arbiters; consequently using the (h, k)-arbiters is better than using the k-arbiters.

An outstanding problem is obtaining an (h, k)-arbiter with minimum size quorums, since the (h, k)-arbiters we have developed are not optimal. Another problem is obtaining a non-dominated (h, k)-arbiter, where non-domination of (h, k)-arbiter can be defined exactly as it is for the k-arbiter [13].

Acknowledgment We would like to thank Dr. Hirofumi Katsuno of NTT for his encouragement and suggestions.

References

[1] Agrawal, D.: Eğecioğlu, Ö., and Abbadi A.E.: "Analysis of quorum-based protocols for distributed (k+1)-exclusion," *Proc. of 1st COCOON, Lecture Notes in Computer Science,* Vol. 959, pp. 161-170 (Aug. 1995).

[2] Baldoni, R. and Ciciani, B.: "Distributed algorithms for multiple entries to a critical section with priority," *Information Processing Letters,* Vol. 50, pp. 165-172 (1994).

[3] Bulgannawar, S. and Vaidya, N.H.: "A distributed K-mutual exclusion algorithm," *Proc. 15th Int. Symp. on Distributed Computing Systems,* pp. 153-160 (June 1995).

[4] Garcia-Molina, H. and Barbara, D.: "How to assign votes in a distributed system," *Journal of the ACM,* Vol. 32, No. 4, pp. 841-860 (1985).

[5] Jiang, J.-R., Huang, S.-T., and Kuo, Y.-C.: "Cohorts structures for fault-tolerant k entries to a critical section," *IEEE Trans. on Computers,* Vol. 48, No. 2, pp. 222-228 (Feb. 1997).

[6] Kakugawa, H., Fujita, S., Yamashita, M., and Ae, T.: "Availability of k-coterie," *IEEE Trans. on Computers,* Vol. 42, No. 5, pp. 553-558 (May 1993).

[7] Kakugawa, H., Fujita, S., Yamashita, M., and Ae, T.: "A distributed k-mutual exclusion algorithm using k-coterie," *Information Processing Letters,* Vol. 49, pp. 213-238 (1994).

[8] Kuo, Y.-C. and Huang, S.-T.: "A simple scheme to construct k-coteries with $O(\sqrt{N})$ uniform quorum sizes," *Information Processing Letters,* Vol. 59, pp. 31-36 (1996).

[9] Kuo, Y.-C. and Huang, S.-T.: "A geometric approach for constructing coteries and k-coteries," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 8, No. 4, pp. 402-411 (Apr. 1997).

[10] Lamport, L.: "Time, clocks, and the ordering of events in a distributed system," *Communications of ACM,* Vol. 21, No. 7, pp. 558-565 (1978).

[11] Maekawa, M.: "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. on Computer Systems,* Vol. 3, No. 2, pp. 145-159 (1985).

[12] Makki, K., Banta, P., Been, K., Pissinou, N., and Park, E.K.: "A token based distributed k mutual exclusion algorithm," *Proc. 4th IEEE Symp. on Parallel and Distributed Processing,* pp. 408-411 (Dec. 1992).

[13] Manabe, Y., Baldoni, R., Raynal, M., and Aoyagi, S.: "k-Arbiter: A safe and general scheme for h-out of-k mutual exclusion," *Theoretical Computer Science,* Vol. 193, No. 1-2, pp. 97-112 (Feb. 1998).

[14] Naimi, M.: "Distributed algorithm for k-entries to critical section based on the directed graphs," *ACM Operating Systems Review,* Vol. 27, No. 4, pp. 67-75 (Oct. 1993).

[15] Neilsen, M.L. and Mizuno, M.: "Nondominated k-coteries for multiple mutual exclusion," *Information Processing Letters,* Vol. 50, pp. 247-252 (1994), Erratum Vol. 60, p. 319 (1996).

[16] Raymond, K.: "A distributed algorithm for multiple entries to a critical section," *Information Processing Letters,* Vol. 30, No. 4, pp. 189-193 (Feb. 1989).

[17] Raynal, M.: "A distributed solution for the k-out of-m resources allocation problem," *Proc. 1st Conf. on Computing and Information, Lecture Notes in Computer Sciences,* Vol. 497, pp. 599-609 (May 1991).

[18] Schlichting, R.D. and Schneider, F.B.: "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. on Computing Systems,* Vol. 1, No. 3, pp. 222-238 (1983).

[19] Singhal, M: "A taxonomy of distributed mutual exclusion," *Journal of Parallel and Distributed Computing,* Vol. 18, No. 1, pp. 94-101 (1993).

[20] Srimani, R.K. and Das, S.R. (eds.): *Distributed Mutual Exclusion Algorithms,* IEEE Computer Society Press (1992).

[21] Srimani, P.K. and Reddy, R.L.N.: "Another distributed algorithm for multiple entries to a critical section," *Information Processing Letters,* Vol. 41, No.1, pp. 51-57 (Jan. 1992).

[22] Yuan, S.M. and Chang, H.K.: "Comments on: Availability of k-coterie," *IEEE Trans. on Computers,* Vol. 43, No. 12, p. 1457 (Dec. 1994).

```
program MutualExclusion; /* Program for process $p$. */
const $Q_{h,k}$ : $(h,k)$-arbiter;
    $p$ : integer; /* Identifier of $p$. */
var $c = 0$ : integer; /* Logical clock. */
  $h = 0$ : integer; /* Request units of shared resource. */
  $In = false$ : boolean; /* True if in the critical section. */
  $Q, K = \phi$ : set of process;
  $x = k$ : integer; /* Number of current tokens. */
  $Que = null$ : queue; /* Priority queue of the requests. */
  $Quelength = 0$ : integer; /* Current length of $Que$. */

When $p$ wants to use $h$ units of shared resource:
begin
    $Q$ :=(one quorum in $Q_{h,k}$);
    $K := \phi$;
    for all $r \in Q$ do send "request($h$, $p$, $c$)" to $r$;
end; /* end of request initiation. */

When "OK" is arrived from $p'$:
begin
  $K := K \bigcup \{p'\}$;
  if $K \neq Q$ then return;
      /* wait for another "OK" */
  $In := true$;
/* Now in the critical section. */
....
/* Exiting from the critical section. */
  $In := false$;
  for all $r \in Q$ do send "release" to $r$
end;

When "request($h'$, $p'$, $c'$)" is arrived from $p'$:
begin
  Insert this request to $Que$
      according to the priority $(p', c')$;
    /* Let this request be $j$-th entry in $Que$. */
  $Que[j].clock := c$,;
  $Que[j].process := p'$;
  $Que[j].units := h'$;
  $Que[j].status$ :='wait';
  $Quelength := Quelength + 1$;
  if ($\sum_{i=1}^{j} Que[i].units \leq k$) and ($x \geq h'$) then
    begin /* Reply permission to this request. */
      send "OK"to $p'$;
      $Que[j].status$ :='ok';
      $x := x - h'$
    end;
    /* Try to cancel "OK" for lower priority requests. */
  $i$ :=maximum integer ($\leq Quelength$)
      that satisfies $\sum_{j=1}^{i} Que[j].units \leq k$;
  if ($i = Quelength$) then return;
  for $j := i + 1$ to $Quelength$ do
    if ($Que[i].status =$'ok') then begin
      send "cancel" to $Que[i].process$;
      $Que[i].status$ :='cancel'
    end
end;

When "cancel" is arrived from $p'$:
begin
  if $In = false$ and $p' \in K$ then begin
    send "cancelled" to $p'$;
    $K := K - \{p'\}$
  end
end;

When "release" is arrived from $p'$:
begin
  Search for the entry in $Que$
      such that $Que[j].process = p'$;
  $x := x + Que[j].units$;
  Delete the entry $Que[j]$ from $Que$;
  $Quelength := Quelength - 1$;
  NewChance
end;

When "cancelled" is arrived from $p'$:
begin
  Search for the entry in $Que$
      such that $Que[j].process = p'$;
  $Que[j].status$ :='wait';
  $x := x + Que[j].units$;
  NewChance
end;

procedure NewChance; /* When tokens are released. */
begin
  for $i := 1$ to $Quelength$ do begin
    if ($Que[i].status =$'wait') then begin
      if ($x < Que[i].units$) then return;
      send "OK" to $Que[i].process$;
      $Que[i].status$ :='ok';
      $x := x - Que[i].units$
    end
  end
end;
```

Figure 1. Distributed h-out of-k mutual exclusion algorithm.