Vol. 37 No. 8 Transactions of Information Processing Society of Japan Aug. 1996

Regular Paper

Local Coteries and a Distributed Resource Allocation Algorithm

HIROTSUGU KAKUGAWA† and MASAFUMI YAMASHITA†

In this paper, we discuss a resource allocation problem in distributed systems. Consider a distributed system consisting of a set of processes and a set of resources of identical type. Each process has access to a (sub)set of the resources. Different processes may have access to different sets of the resources. Each resource must be accessed in a mutually exclusive manner, and processes are allowed to request more than one resource at a time. Since all resources are of identical type, a process requesting k resources does not insist on k particular resources. However, once a resource has been allocated to a process, it cannot be allocated to another process until it is released. The mutual exclusion and k-mutual exclusion problems can be considered as special cases of the resource allocation problem. We first introduce a new class of quorum sets named *local coteries* as an extension of coteries, to take advantages of the fact that, in general, resources are not shared by all processes. Then, we propose a resource allocation algorithm, using a local coterie, that is both deadlock- and starvation-free. This algorithm allows resources requested by two processes to be allocated without any interference.

1. Introduction

A distributed system can be viewed as a set of processes that share many types of resources, such as processors, memory cells, buses, and printers, many of which must be accessed in a mutually exclusive manner; that is to say, they can be accessed by at most one process at a time. Therefore the mutual exclusion problem[10] — that of guaranteeing mutually exclusive access to a resource — is considered to be a basic problem in distributed systems, and has been investigated extensively. If there are k resources (e.g., printers) with the same function (e.g., printing text files), then a process wishing to use the function wants to access an arbitrary one of the k resources, and it may not need to insist on a particular one. As a natural extension of the mutual exclusion problem, the problem of resolving this type of conflict, which is called the k-mutual exclusion problem, has recently attracted increasing attention from researchers[1],[2],[7],[8],[12],[15],[18].

In the conflict-resolution problems mentioned above, every process can access each of the k resources. In real distributed systems, however, processes typically can access only subsets of the resources. This paper will discuss the problem of resolving the above type of conflict, which we call the (*distributed*) *resource allocation problem.*

To provide the reader with a more concrete image, we would like to introduce a small example. Suppose that there are three processes, a, b and c, and two resources, x and y. Suppose that a, b, and c respectively can access x, x and y, and y (**Fig. 1**). If b requests a printer when a is printing on x, then we want to allocate y to b. If b requests a printer when both a and c are printing, we want to allocate either x or y as soon as either one becomes available. If b requests two printers, we want to allocate x and y, so we must wait for both of them to become available. This problem is called the *drinking philosophers problem* in Chandy and Misra[4] and has been extensible investigated in the last decade, for example,[3],[5]. In the resource-sharing model adopted in these papers, a resource is shared by only a pair of processes. That is, its sharing relation is described by a *conflict graph*: a node represents a process, and an edge represents a resource that is shared by two processes incident to the edge. In this paper, we discuss a more general sharing model such that edges are hyper edges; that is, resources can be shared by more than two processes.

One may argue the following simple solution of the problem: Execute a mutual exclusion algorithm ALG(r) for each resource r, and have processes specify the name of one of the resources when issuing a resource request. But it is not a successful solution from our viewpoint, because, for example, in the second situation of the above example, if b requests printer x, then it must wait until x becomes available, even if y becomes available considerably earlier than

† Faculty of Engineering, Hiroshima University

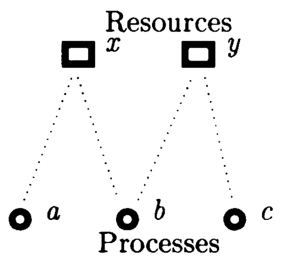


Fig. 1 An example of resource sharing

x. If we allowed b to request both x and y, we would inevitably allow processes to request all resources simultaneously (i.e., a process would access the one that becomes available first and fail to cancel the requests for the others); the resulting algorithm would be extremely inefficient.

Raynal proposed the k-out-of-M resource allocation problem in Ref. 16). He considered a problem of allocating any number k of M resources. In his algorithm, a process sends messages to every other process; this it is not efficient. Baldoni also proposed an algorithm for the problem[2]. His algorithm uses a *k-coterie*☆ to reduce the message complexity. These algorithms only guarantee that the number of allocated resources is at most M at any time, and a process cannot know the *names* of allocated resources. A general resource-sharing model for anonymous resources is investigated by Miyamoto[13].

This paper proposes a distributed algorithm for solving the resource allocation problem we informally introduced above. (A formal definition will be given in the next section.) The algorithm makes it possible for the processes to keep track of accesses to resource names, unlike the algorithms in Refs. 2), 13), 14), 16). In some real situations, this function is definitely very useful: Suppose, for example, that there are two printers on the first and second floors, respectively, and that a file is output on one of them. Then we may want to know from which one it is output.

Maekawa used a structure of processes called a *coterie*, which was proposed by Garcia-Molina and Barbara[6], to design a distributed algorithm for the mutual exclusion problem. This algorithm was shown in Ref. 11) to be efficient in message complexity. Since then, the quorum-based approach, which Maekawa adopted, is considered to be a promising way of designing algorithms for conflict resolution problems. Our approach is similar; we introduce a new process structure called a *local coterie*, which is an extension of the coterie. Finally, we show the correctness of our resource-allocation algorithm.

☆ Kakugawa, et al. also proposed a concept of k-coterie[7],[8], that is different from the one proposed by Baldoni[2].

The organization of this paper is as follows. In Section 2, we give a model of the distributed systems assumed in this paper and define the distributed resource allocation problem. In Section 3, we introduce local coteries and give a simple algorithm for constructing a local coterie. In Section 4, we propose a distributed resource allocation algorithm that uses local coteries. The correctness of the algorithm is shown in Section 5. In Section 6, we analyze the message complexity of the proposed algorithm. Finally, we summarize this paper in Section 7.

2. The Model

The Distributed System Model: A distributed system consists of n processes $U = \{u_1, u_2, \ldots, u_n\}$, bidirectional communication links each connecting two processes, and m resources $R = \{r_1, r_2, \ldots, r_m\}$ shared by processes. The network topology is assumed to be complete; any two processes are connected by a link.

Each process has a unique identifier (process ID) selected from an integer set $\{1, 2, \ldots, n\}$. The processing speeds of processes may be different, and may even vary during the execution of a program. However, every process is guaranteed to execute the next instruction within a finite time, unless it has been terminated. Each process has its own local clock, but clocks may indicate different times, and no processes can tell the global time.

The only mechanism provided in the system for information exchange between processes is point-to-point message passing. More precisely, each process has a message queue of infinite length, which stores arriving messages. Each message is delivered in a finite time. The delivery delay is unpredictable, but the order of messages is kept unchanged during the delivery. Finally, we assume that both processes and links are error-free.

Processes $u \in U$ are allowed to use some of the resources $r \in R$. By the function $\alpha : U \to 2^R$, we denote the following relation; for any $u \in U$,

$\alpha(u) = \{r \in R \mid u \text{ has access to } r\} \subseteq R.$

In this paper, we assume that $\alpha(v) \neq \emptyset$ for any $v \in U$. When V is a set of processes, $\alpha(V)$ denotes $\cup_{v \in V} \alpha(v)$. The triple (U, R, α) is called the *sharing structure* of the system.

Define a *configuration* c of the distributed system as a tuple of the states of all processes and communication links. For any $u \in U$, $\rho_u(c)$ denotes the set of resources that are being accessed by u when the system is in configuration c. A computation π of the system can be described by a (possibly infinite) sequence of configurations starting from an initial configuration. Note that because of the asynchrony of the system, the computation is not determined uniquely, in general, even if an initial configuration is given.

The Resource Allocation Problem: Consider a distributed system such that each process consists of an infinite cycle of a computation phase followed by a resource access phase. The computation phase does not contain resource access instructions, whereas the resource access phase consists of a series of resource access instructions. The latter phase starts with a resource request instruction for requesting a number of resources, and terminates with a resource release instruction for releasing all resources it is accessing. Let $S = (U, R, \alpha)$ be its sharing structure. Each time the resource access phase is executed, the number of resources a process u requests can change between 1 and $|\alpha(u)|$.

The *resource allocation problem* is the problem of implementing the resource request and release instructions in such a way that whenever a process u requests k $(\leq |\alpha(u)|)$ resources, eventually k resources are allocated to u. Furthermore, as the restriction arising from the sharing structure, any computation $\pi = c_0, c_1, \ldots, c_i, \ldots$ of the resulting distributed system must satisfy the following two conditions:

Allocation Validity: For any configuration c_i and any process $v \in U$,

$$\rho_v(c_i) \subseteq \alpha(v).$$

Mutual Exclusion: For any configuration c_i and any two processes $u, v \in U$,

$$\rho_u(c_i) \cap \rho_v(c_i) = \emptyset.$$

Allocation Validity guarantees that a process u only accesses resources to which it has access, and Mutual Exclusion guarantees that every resource is allocated to at most one process at a time.

3. Local Coteries

Garcia-Molina and Barbara[6] introduced the concept of a *coterie*, a main motivation of which is to design a mutual exclusion algorithm that is both efficient in message complexity and robust with respect to network failures. A set $Q = \{q_1, q_2, \cdots, q_{|Q|}\} \subseteq 2^U$ is called a *coterie* under $U = \{u_1, u_2, \ldots, u_n\}$ if all of the following conditions hold:

- Non-emptiness: $\forall q \in Q[q \neq \emptyset]$
- Intersection property: $\forall q, r \in Q[q \cap r \neq \emptyset]$
- Minimality: $\forall q, r \in Q[q \not\subseteq r]$

An element of a coterie is called a *quorum*. Roughly, a coterie is used to solve the mutual exclusion problem as follows: Initially, each process u holds one token called "permission (by u)". Select a coterie Q and let each process use Q. A process u that wants a resource (in other words, one that wants to enter the critical section) sends request messages to all processes in a quorum $q \in Q$. Upon receiving u's request, process v sends its permission to u if it holds it. Otherwise, when the permission returns, it sends to one of the processes that are waiting for it. (How to select the one from the waiting processes is the key to make the resulting algorithm deadlock- and starvation-free, and we would like to encourage readers to check, for example, Refs. 11), 17).) A process u can access the resource if it receives the permission from each of the processes in q. When u releases the resources, it returns the permissions. The intersection property guarantees that at most one process can access the resource.

Unlike the mutual exclusion problem, the resource allocation problem, in general, involves cases in which resources are shared by different sets of processes. Consider a case in which two processes u and v do not share a resource. Then it is a natural requirement that their requests be interference-free. (This may or may not be possible, depending on the remaining part of the sharing structure.) As long as the same set of quorums is associated with u and v, like the above mutual exclusion algorithm, the interference inevitably occurs, because of the intersection property.

To take account of the sharing structure, we associate (possibly different) sets of quorums with each process. We call a set $\{Q_u \mid u \in U\}$ a *local coterie* with respect to a sharing structure (U, R, α) if all of the following conditions hold:

- Non-emptiness: $\forall u \in U[Q_u \neq \emptyset]$.

- Intersection property: $\forall u, v \in U[\alpha(u) \cap \alpha(v) \neq \emptyset \Rightarrow \forall q \in Q_u, \forall r \in Q_v[q \cap r \neq \emptyset]]$.
- Minimality: $\forall u \in U, \forall q, r \in Q_u[q \not\subseteq r]$.

Note that the definition of a local coterie includes that of a coterie as a special case when $|R| = 1$ and $\alpha(u) = R$ for all $u \in U$.

When two processes u and v do not share resources, there can be non-intersecting quorums of u and v, and consequently resource allocation for u and v can be performed independently. This is the motivation for introducing local coteries.

Consider, for example, the following sharing structure (U, R, α), where $U = \{u_1, u_2, u_3, u_4\}$, $R = \{r_1, r_2, r_3, r_4, r_5\}$, and $\alpha(u_i) = \{r_i, r_{i+1}\}$ for all $1 \leq i \leq 4$. Then, it is easy to check that the following quorum sets $\{Q_u \mid u \in U\}$ is a local coterie with respect to (U, R, α):

- $Q_{u_1} = \{\{u_1, u_2\}\}$,
- $Q_{u_j} = \{\{u_{j-1}, u_j, u_{j+1}\}\}$ for each $j = 2, 3$,
- $Q_{u_4} = \{\{u_3, u_4\}\}$.

It should be noted that Q_{u_1} and Q_{u_4} do not contain quorums intersecting each other, which reflects the fact that u_1 and u_4 do not share resources, that is, $\alpha(u_1) \cap \alpha(u_4) = \emptyset$, and therefore, the resource allocation for u_1 and u_4 can be treated independently. (This quorum set is generated by an algorithm described below.)

First, for any sharing structure (U, R, α), we show that there certainly exists a local coterie with respect to (U, R, α).

```
Algorithm LocalCoterie$(U, R, \alpha)$;
begin
    $G := MakeBipartiteGraph(U, R, \alpha)$;
    for each $r$ in $R$
        $N(r) := AllAdjacentNodes(r, G)$;
    for each $u$ in $U$
        $q_u := \bigcup_{r \in \alpha(u)} N(r)$;
    for each $u$ in $U$
        $Q_u := \{q_u\}$;
    (* $Q_u$ is the quorum set consisting
        only of one quorum. *)
    return $\{Q_u \mid u \in U\}$
end.                                          □
```

Function $MakeBipartiteGraph(U, R, \alpha)$ constructs a bipartite graph $G = (U \cup R, E)$ with partite sets U and R, where $(v, r) \in E$ iff $r \in \alpha(v)$. Function $AllAdjacentNodes(r, G)$ constructs the set $N(r)$ of all processes that are adjacent to r in G, i.e., $N(r) = \{u \in U \mid r \in \alpha(u)\}$.

Theorem 1
Algorithm *LocalCoterie*(U, R, α) computes a local coterie with respect to a given sharing structure (U, R, α) in $O(|U|^2 \cdot |R|)$ time.

Proof: Non-emptiness holds, because for each $v \in U$, there is a resource $r \in \alpha(q_v)$, by assumption, and $v \in N(r)$, which implies that $\{v\} \subseteq q_v$.

Minimality trivially holds, since the quorum set Q_v contains only one quorum for each process $v \in U$.

Assume that the intersection property does not hold. Let v_1 and v_2 $(v_1 \neq v_2)$ be two processes such that $(\alpha(v_1) \cap \alpha(v_2) \neq \emptyset) \wedge (q_{v_1} \cap q_{v_2} = \emptyset)$, where $q_{v_i} \in Q_{v_i}$ for $i = 1, 2$. Let r be a resource in $\alpha(v_1) \cap \alpha(v_2)$. Because $v_1, v_2 \in N(r)$, we have $\{v_1, v_2\} \subseteq q_{v_1} \cap q_{v_2}$; This is a contradiction.

Since G is a bipartite graph, the maximum number of its edges is $|U| \cdot |R|$. Thus, G can be constructed in $O(|U| \cdot |R|)$ time. $N(r)$ can be computed from G in $O(|U|)$ time. $\bigcup_{r \in \alpha(u)} N(r)$ can be computed in $O(|U| \cdot |R|)$. Thus, the execution time is $O(|U|^2 \cdot |R|)$. □

Corollary 1 For any sharing structure (U, R, α), there exists a local coterie C with respect to the sharing structure (U, R, α). □

4. The Distributed Resource Allocation Algorithm

We are now ready to introduce our algorithm for solving the distributed resource allocation problem. We first give an outline of the algorithm, and then describe it in detail.

We say that a resource r is *available* to u if $r \in \alpha(u)$ and r is currently free. On the other hand, when we say that r is *accessible* by u, it simply means that $r \in \alpha(u)$.

Together, the processes maintain a distributed database, which keeps pairs each consisting of a process and a resource it is currently accessing. A process u wishing to access k resources sends a query asking whether or not there are k resources available to u. If the answer is yes, then the k resources are allocated to u.

An essential contribution of this paper is to use local coteries to implement the distributed database efficiently. Let $\{Q_u \mid u \in U\}$ be a local coterie, where Q_u is the quorum set associated with process u. Then the outline of the algorithm is as follows.

In our algorithm, a process v is (partially) responsible for resources r that are accessible by a process w, if v occurs as an element of a quorum in Q_w. When w is accessing r, as explained below, w has obtained permission from every process v in a quorum q in Q_w. Process v, on the other hand, memorizes the fact that w is accessing r as a part of its database.

A process u wishing to access k resources selects an arbitrary quorum $q \in Q_u$, and sends a query message ⟨QUERY⟩ to every process v in q. A process v receiving the query ⟨QUERY⟩ sends back the names of resources available to u. Upon receiving the list of available resource names from every process $v \in q$, u arbitrarily selects k resources among those common to all lists and sends a lock message ⟨LOCK⟩ with the k names to every process v in q to let it update the current states of the k resources. When u releases the k resources, it sends an unlock message ⟨UNLOCK⟩ with the k names to every process v in q to let it free the states of the resources.

The above description of the algorithm does not explain how to avoid deadlocks and starvations or how to treat cases in which u cannot find k resources available to u. Moreover, in order for the algorithm to work correctly, the query step must be carried out in a mutually exclusive manner. Nevertheless, we would like to observe that if u decides to access a set of resources R', then R' is currently available to u (i.e., u has access to R' and no resource in R' is used by some process), from the definition of a local coterie. To avoid deadlocks, we use the messages ⟨PREEMPT⟩ and ⟨RETURN⟩ to preempt and return exclusive access to a quorum.

The algorithm assumes that each process u maintains the following local variables. For convenience of explanation, as in the above rough explanation, define

$$S_u = \{w \mid u \in q \text{ for some } q \in Q_w\},$$
$$R_u = \bigcup_{w \in S_u} \alpha(w).$$

Intuitively, S_u is a set of processes w such that at least one of w's quorum includes u. The set R_u is a set of all resources accessible by each process in S_u and u (partially) manages the allocation of resources in R_u.

- C_u — The current logical time at u. It is initially 0 and is automatically incremented[☆].
- D_u — The array that memorizes whether or not r is allocated to a process for each $r \in R_u$. More precisely, $D_u(r) = (v, t)$ if r is locked by a ⟨LOCK⟩ message with timestamp t issued by v. Note that this occurs only when v, wishing to access r, selected a quorum $q \in Q_v$ such that $u \in q$. Otherwise, $D_u(r) = (\perp, t)$, where t is the timestamp attached to the latest ⟨UNLOCK⟩ message received by u. Initially, $D_u = (\perp, 0)$ for all $r \in R_u$.
- W_u — The name of process to which u sends the current states of resources. (u is waiting for a ⟨LOCK⟩ message as a reply from this process.) If u is not waiting for a reply message, $W_u = \perp$.
- T_u — The timestamp attached to the ⟨QUERY⟩ message that the process held in W_u issued. $T_u = \perp$ if $W_u = \perp$.
- X_u — The priority queue to hold ⟨QUERY⟩ messages waiting at u for their turns. They are sorted in the order of their timestamps.

We describe our algorithm *AllocResource* in an event-driven form.

Algorithm *AllocResource*;

Let $\{Q_u\}$ be the local coterie used in the algorithm. For simplicity of description, the parameters of a message may be omitted, and we may write, for example, ⟨LOCK⟩ instead of ⟨LOCK, u, C_u, G_u⟩.

(1) **When a process u wishes to access $k(\leq |\alpha(u)|)$ resources:**
It arbitrarily selects a quorum $q \in Q_u$, and sends a ⟨QUERY, u, C_u⟩ to every process in q[☆☆]. Recall that C_u is the current logical time at u and is used to timestamp the message. Process u waits until both of the following two conditions hold:
  - It has received at least one messages of type ⟨RESPONSE, v, D_v⟩ from each process $v \in q$. Note that v sends a ⟨RESPONSE, v, D_v⟩ message carrying the latest version of D_v as soon as D_v is updated, even if it has sent an older version to u (see Case 7). Note also that u does not need to store old versions. It simply discards them and holds the latest one (see Case 3).
  - A_u contains at least k resources, where $A_u \subseteq \alpha(u)$ is a set of resources r satisfying $D_v(r) = (\perp, t_v)$ for all

[☆] By means of a standard technique that uses unique process IDs, events occurring in the system are fully ordered according to the logical time[9].

[☆☆] The number k of requesting resources is not a parameter of a ⟨QUERY⟩ message.

$v \in q$. Recall that every D_v contains the states of all resources in $\alpha(u)$ from the view of v. Intuitively, A_u is the set of resources currently available to u, as we will show in the next section.

If both of the above conditions hold, u arbitrarily selects a set of k resources from A_u, say R', sends a $\langle$LOCK$, u, C_u, R'\rangle$ message to every process $v \in q$, and accesses R'.

(2) **When a process u releases the set G_u of resources:**
It sends an $\langle$UNLOCK$, u, C_u, G_u\rangle$ message to every process $v \in q$.

(3) **When a process u receives a $\langle$RESPONSE$, v, D_v\rangle$ message from a process v:**
It stores D_v. If it has received an older version of D_v, it discards it and stores the latest one. Because messages are assumed to be delivered in order, u always holds the latest version among of the versions received so far.

(4) **When a process v receives a $\langle$QUERY$, u, t\rangle$ from a process u:**
If $W_v = \perp$, that is, if process v is not waiting for a $\langle$LOCK$\rangle$ message from another process, it sends a $\langle$RESPONSE$, v, D_v\rangle$ message to u, and sets $W_v := u$ and $T_v := t$. Recall that t is the logical time at the process u where the $\langle$QUERY$\rangle$ message was issued (see Case 1). Otherwise, $W_v = w$ for some process $w \in U$, that is, w waits for the two conditions in Case 1 to hold. If $T_v < t$, that is, if w has higher priority (since T_v is the timestamp attached to w's $\langle$QUERY$\rangle$), v stores $\langle$QUERY$, u, t\rangle$ to queue X_v. Otherwise, if $T_v > t$, u has the higher priority. Then, in order to preempt the right to lock resources, that v gave to w, v sends a $\langle$PREEMPT$, v\rangle$ to w, and waits for w to reply either with a $\langle$RETURN$\rangle$ or a $\langle$LOCK$\rangle$ message (see Cases 1 and 8), after storing the $\langle$QUERY$\rangle$ messages issued by u and w to X_v. When v again needs to send a $\langle$PREEMPT$\rangle$ to w while waiting for a reply from w, v ignores it.

(5) **When a process v receives a $\langle$RETURN$, w\rangle$ message from a process w:**
It takes the $\langle$QUERY$, x, t\rangle$ message from the top of queue X_v. This is the $\langle$QUERY$\rangle$ message that has the highest priority. Then, v sends a $\langle$RESPONSE$, v, D_v\rangle$ to x, and sets $W_v := x$ and $T_v := t$.

(6) **When a process v receives a $\langle$LOCK$, w, t, G_w\rangle$ message from a process w:**
It updates its data D_v by setting $D_v(r) := (w, t)$, for each $r \in G_w$. Then it continues (the algorithm fragment for) Case 5 if X_v is not empty.

(7) **When a process v receives an $\langle$UNLOCK$, w, t, G_w\rangle$ message from a process w:**
It updates its data D_v; it sets $D_v(r) := (\perp, t)$, for each $r \in G_w$. If $W_v \neq \perp$, it sends a $\langle$RESPONSE$, v, D_v\rangle$ message to W_v. Otherwise, it continues Case 5 if X_v is not empty.

(8) **When a process w receives a $\langle$PREEMPT$, v\rangle$ message from a process v:**
If it has sent back an $\langle$UNLOCK$\rangle$ message to v, it simply ignores the $\langle$PREEMPT$\rangle$ message. Otherwise, it sends back a $\langle$RETURN$, w\rangle$ message, and then discards the copy of D_v at w that was previously sent by a $\langle$RESPONSE$, v, D_v\rangle$ message from v. (Recall that w keeps D_v unchanged for each process v.) Then, w waits for another $\langle$RESPONSE$\rangle$ message from v. □

Although a $\langle$RESPONSE$\rangle$ message carries all of D_v in the above description of *AllocResource*, it is enough to carry the data on $\alpha(u)$ in D_v, since a process u will use information on $\alpha(u)$ in D_v.

5. Proof of Correctness

In this section, we show the correctness of *AllocResource*, provided that processes accessing resources release them within a finite time.

Since a process u selects the resources it accesses from the candidate set A_u, which is a subset of $\alpha(u)$, the following theorem holds.

Theorem 2 Algorithm *AllocResource* guarantees Allocation Validity. □

Before we proceed to the remaining properties, recall that a process u requesting k resources arbitrarily selects k resources from A_u determined from D_v's for $v \in q \in Q_u$. It then sends a $\langle$LOCK$\rangle$ message carrying the names of k resources to every v, after which it is free to access the k chosen resources. Process v, on the other hand, updates D_v in response to the ⟨LOCK⟩ message. If two processes that share resources received D_v's simultaneously, they could select the same resources and access them simultaneously. Our algorithm guarantees that this situation never occurs. We introduce the notion of a *Q-region* to prove this formally.

A process u requesting k resources sends a ⟨QUERY⟩ message to every member v of a quorum $q \in Q_u$, and collects D_v's until the two conditions of Case 1 hold. If a ⟨PREEMPT⟩ message from $w \in q$ arrives in the meanwhile, process u discards D_w and waits for a new D_w. Recall that receiving a D_v from every $v \in q$ is a necessary but not sufficient condition. We say that u is in the *Q-region* if u has received a D_v from every $v \in q$, but has neither sent a ⟨LOCK⟩ message nor received a ⟨PREEMPT⟩ message since then.

Lemma 1 Let u and v be any two processes such that $\alpha(u) \cap \alpha(v) \neq \emptyset$. Then u and v are never in their Q-regions simultaneously.

Proof: Assume that there exist two processes u and v such that $\alpha(u) \cap \alpha(v) \neq \emptyset$ and u and v are in their Q-regions at the same time. Let w be a process such that w is in both the quorums chosen by u and v. Note that such a w exists, since $\alpha(u) \cap \alpha(v) \neq \emptyset$. Without loss of generality, assume that w sent a ⟨RESPONSE⟩ message to u first. From the definition of the algorithm, w extracts the request from v after sending a ⟨RESPONSE⟩ message to u. By assumption, w sent a ⟨RESPONSE⟩ message to v before a ⟨LOCK⟩ or ⟨RETURN⟩ message was sent from u. This action contradicts the definition of the algorithm. □

Suppose that a resource r has been allocated. If no v knew this fact when it sent D_v for the first time to u, A_u could include r. In this case, r may be allocated to more than one process, since the candidate set A_u is determined from the D_v's. The next lemma guarantees that this situation never occurs.

Lemma 2 Let u and v be any two processes such that $r \in \alpha(u) \cap \alpha(v) \neq \emptyset$. Assume that r has been allocated to u, and that v is now in its Q-region. Further, assume that u used quorum $q_u \in Q_u$ for resource request and that v is using quorum $q_v \in Q_v$. Then for any $w \in q_u \cap q_v$, $D_w(r) = (u, t)$ for some t.

Proof: From the definition of a local coterie, $q_u \cap q_v \neq \emptyset$. Since u is accessing a resource r, it sent a ⟨LOCK⟩ message to every process in q_u when it exited from the Q-region, and it then started accessing r. Every $w \in q_u \cap q_v$ sends a ⟨RESPONSE⟩ message to v after it receives a ⟨LOCK⟩ message from u, because v is in the Q-region.

When w receives a ⟨LOCK⟩ message from u, it updates its local database so that r is allocated to u, along with its allocation time. When w sends a ⟨RESPONSE⟩ message to v, w knows that r is already allocated. Thus, $D_w(r) = (u, t)$ for some t. □

Theorem 3 The algorithm *AllocResource* guarantees mutual exclusion.

Proof: Assume that a resource $r \in \alpha(u) \cap \alpha(v)$ is allocated to both u and v simultaneously. The proof is by induction. The mutual exclusion condition holds at the initial state of the system, since no resources are allocated to processes. From lemma 1, no two processes sharing resources are in their Q-regions simultaneously. Without loss of generality, we assume that u leaves its Q-region first by sending a ⟨LOCK⟩ message to allocate r to u. Then, v can enter its Q-region only after all processes in $q_u \cap q_v$ receive a ⟨LOCK⟩ message from u, where $q_u \in Q_u$ ($q_v \in Q_v$) is the quorum that u (v) chooses for response request. Since u and v share resources, $q_u \cap q_v$ is not empty. Let w be any process in $q_u \cap q_v$. Then, w updates its database so that $D_w(r) = (u, t_u)$ holds for some t_u when it receives the ⟨LOCK⟩ message from u. Hence, every ⟨RESPONSE⟩ message sent to u from w contains $D_w(r) = (u, t_u)$. Therefore v cannot choose r; this is a contradiction. □

Theorem 4 Algorithm *AllocResource* is deadlock free.

Proof: Since processes request all necessary resources when the resource access phase starts, we do not consider deadlocks caused by nested requests. We consider the deadlocks at the query step.

Assume that a deadlock occurs. Since the number of processes is finite, there exists a time such that the number of processes that are deadlocked does not increase afterwards. We consider what will happen. Although there may exist processes that do not send and/or receive messages in general, we can assume without loss of generality that there are no such processes.

Let $V \subseteq U$ be the set of processes that are deadlocked, and assume that $u \in V$ is the process whose timestamp attached to the ⟨QUERY⟩

message is the smallest (i.e., it has the highest priority) among V. The ⟨QUERY⟩ message sent by u will reach every process in a quorum $q \in Q_u$ in a finite time. Since the logical clock monotonically increases, the timestamp of u's ⟨QUERY⟩ message will become the smallest among all processes. From the definition of the algorithm, each process v in q behaves as follows. If v sent a ⟨RESPONSE⟩ message to a process $w \in U$ but has not received the corresponding ⟨LOCK⟩ message, then v sends a ⟨PREEMPT⟩ message to w to switch the query right to u. If v receives a ⟨RETURN⟩ message from w, it will send a ⟨RESPONSE⟩ message to u. Otherwise, it will send a ⟨RESPONSE⟩ message to u, when w returns a ⟨LOCK⟩ message, since u's ⟨QUERY⟩ message has the highest priority. On the other hand, processes that share resources with u cannot be in their Q-region, and hence, resources are not allocated to them. Therefore, within a finite time, enough number of resources in $\alpha(u)$ become free and the request by u will be satisfied within a finite time; This is a contradiction. □

Theorem 5 Algorithm *AllocResource* is starvation-free.

Proof: Assume that starvation occurs, and let u be the starved process that has the smallest timestamp (i.e., the highest priority). Without loss of generality, we ignore processes that never request resources and whose timestamps do not increase, because they have no interaction with other processes.

Let $q \in Q_u$ be a quorum chosen by u and let v_i be any process in q. If v_i has sent a ⟨RESPONSE⟩ message to u, then v_i never sends a ⟨PREEMPT⟩ message, because u has the highest priority. Thus, v_i has not sent a ⟨RESPONSE⟩ message to u. But the request by a ⟨QUERY⟩ is enqueued in X_{v_i}, according to the timestamps, and every request (if any) that has higher priority than u's in X_{v_i} is a request by a non-starved process. Thus, such requests are met and the request by u eventually comes to the top of the queue. Every process using resources eventually releases them, and lists of free resources are sent to u by a ⟨RESPONSE⟩ message. □

We can conclude that the algorithm *AllocResource* correctly solves the resource allocation problem.

Theorem 6 Algorithm *AllocResource* solves the resource allocation problem. □

6. Message complexity

In this section, we analyze the message complexity of the proposed algorithm. In the best case, the messages ⟨QUERY⟩, ⟨RESPONSE⟩, ⟨LOCK⟩, and ⟨UNLOCK⟩ are used for each process in a quorum. Thus, the message complexity in the best case is $4|q|$, where q is the smallest quorum.

The worst case happens in the following situation: A process u sends a ⟨QUERY⟩ message to every process v_i in a quorum q to use $|\alpha(u)|$ resources. When a ⟨QUERY⟩ message arrives at v_i, v_i has sent a ⟨RESPONSE⟩ to some process w_i and is waiting for a ⟨LOCK⟩ message. The priority of u is higher, and v_i sends a ⟨PREEMPT⟩ message to w_i. Then, w_i sends a ⟨RETURN⟩ message to v_i and v_i sends a ⟨RESPONSE⟩ message to u. These messages (plus a ⟨LOCK⟩ message) are necessary for u to be in its Q-region.

A requesting process u may not find enough resources through the responses it receives. Assume that u cannot find any resources. That is, assume that each v_i has allocated resources to some processes. If these processes are using one resource per process and they release their resources one after another, each time v_i receives an ⟨UNLOCK⟩ message, v_i sends the latest D_{v_i} embedded in a ⟨RESPONSE⟩ message to u. Because u is requesting $|\alpha(u)|$ resources, v_i sends a ⟨RESPONSE⟩ message $|\alpha(u)|$ times.

When u unlocks the resources, it sends an ⟨UNLOCK⟩ message to v_i and then v_i sends a ⟨RESPONSE⟩ message (possibly) to w_i.

Since the above situation can happen for each process in a quorum, the message complexity in the worst case is $(7 + |\alpha(u)|)|q|$, where $q \in Q_u$.

7. Discussion

In this paper, we have discussed the resource allocation problem, and proposed a distributed algorithm. Unlike for other conflict resolution problems such as the mutual exclusion and the k-mutual exclusion problems, we consider cases in which processes may have access rights to different sets of resources. To take account of the resource-sharing relation of the system, we have introduced a new concept called "local coterie."

The number of messages that need to be exchanged per resource request can be shown to be $4|q|$ in the best case and $(7 + |\alpha(u)|)|q|$ in the worst case, where $|q|$ is the quorum size. In cases where each resource is shared by a small number of processes, since the quorum size $|q|$ can be small, our algorithm is suitable.

Manabe and Aoyagi proposed a distributed resource allocation algorithm for anonymous resources[12]. Their algorithm uses a k-coterie[2),12)] and runs in $O((2h+3)|q|+3)$ message complexity, where h is the number of resources a process requests and $|q|$ is the size of a quorum. A drawback of the k-coterie is that $|q|$ becomes large when the number k of resources increases. For example, Baldoni[2] proposed a method of constructing a k-coterie whose quorum size is $O(n^{k/k+1})$, where n is the number of processes. When k becomes large, the quorum size approaches $O(n)$.

On the other hand, in our algorithm, we can simply use, for example, a coterie based on a finite projective plane[11], whose quorum size is approximately $\sqrt{n}$. We may even be able to find a better local coterie that takes advantage of the sharing structure under consideration. The basic idea of a local coterie is that processes that do not share any resources should not interfere with each other, and quorums are designed not to intersect as possible.

When $|\alpha(u)|$ is large, our algorithm is also less efficient in the worst case. But the worst case described in section 6 seems not to occur very often. To evaluate the average performance, simulation is necessary.

We can improve the algorithm by modifying the proposed algorithm to choose "optimal" resources. For instance, consider the following situation: a secretary working on the 6th floor wants to print a file. She has three accessible printers, LP4, LP5, and LP6, located on the 4th, 5th, and 6th floor, respectively. The "optimal" resource would be LP6, since it is on the same floor. The problem can be solved by defining a priority for each resource and selecting the free resource whose priority is the highest. Note that different users can assign different priorities to the resources. It is easy to include the resource selection strategy for each process in the proposed algorithm.

Finally, we would like to mention some topics for future work. It is possible to use a singleton coterie, namely, $\{\{u\}\}$, for some process u as a local coterie to minimize the message complexity. This is by no means a good selection. Our algorithm has the general advantage of the quorum-based approach of being robust with respect to process and/or link failures; as long as at least one quorum "survives," there is a possibility that resource allocation can be continued. However if a singleton coterie $\{\{u\}\}$ is adopted, since a failure on u is fatal, the robustness of the algorithm fails. Discussion of the fault-tolerance aspect of this algorithm, in connection with a criterion for good local coteries, is left as a topic for future work.

References

1) Baldoni, R.: Mutual Exclusion in Distributed Systems, PhD Thesis, Universita di Roma "La Sapienza" (1994).

2) Baldoni, R.: An $O(N^{M/(M+1)})$ Distributed Algorithm for the k-out of-M Resources Allocation Problem, *The 14th International Conference on Distributed Computing Systems*, pp.81–88 (1994).

3) Bar-Ilan, J. and Peleg, D.: Distributed Resource Allocation Algorithms, *Lecture Notes in Computer Science*, Vol.647 (WDAG '92), pp.276–291 (1992).

4) Chandy, K.M. and Misra, J.: The Drinking Philosophers Problem, *ACM Transactions on Programming Languages and Systems*, Vol.6, No.4, pp.632–646 (1984).

5) Choy, M. and Singh, A.K.: Efficient Fault-Tolerant Algorithms for Distributed Resource Allocation, *ACM Transactions on Programming Languages and Systems*, Vol.17, No.3, pp.535–559 (1995).

6) 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).

7) Kakugawa, H., Fujita, S., Yamashita, M. and Ae, T.: Availability of k-Coterie, *IEEE Transactions on Computers*, Vol.42, No.5, pp. 553–558 (1993).

8) Kakugawa, H., Fujita, S., Yamashita, M. and Ae, T.: A Distributed k-Mutual Exclusion Algorithm Using k-Coterie, *Information Processing Letters* (1994).

9) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol.21, No.7, pp.558–565 (1978).

10) Lamport, L. and Lynch, N.: Distributed Computing: Models and Methods, *Handbook of Theoretical Computer Science, Vol.B: Formal Methods and Semantics*, van Leeuwen, J. (Ed.), The MIT Press/Elsevier, Cambridge/Amsterdam, pp.1157–1200 (1990).

11) Maekawa, M.: A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Transactions on Computer Systems*, Vol.3, No.2, pp.145–159 (1985).

12) Manabe, Y. and Aoyagi, S.: A Distributed k-mutual Exclusion Algorithm Using k-coterie,

COMP 93-43 (1993).

13) Miyamoto, H.: A Study on Quorum Based Approach for Splvong the Anonymous Resource Conflict Resolution Problem, Master's Thesis, Hiroshima University (1994).

14) Miyamoto, H., Kakugawa, H. and Yamashita, M.: An Extended Distributed k-Mutual Exclusion, Information Processing Society Japan, SIG Algorithm Record 34-2 (1993). (in Japanese)

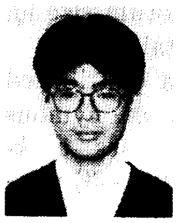
15) Raymond, K.: A Distributed Algorithm for Multiple Entries to a Critical Section, *Information Processing Letters*, Vol.30, pp.189–193 (1989).

16) Raynal, M.: A Distributed Solution to the k-out of-M Resources Allocation Problem, *Lecture Notes in Computer Science*, Vol.497, Springer-Verlag, pp.599–609 (1991).

17) Singhal, M. and Shivaratri, N.G.: *Advanced Concepts in Operating Systems - Distributed, Database, and Multiprocessor Operating Systems*, McGraw-Hill (1994).

18) Srimani, P.K. and Reddy, R.L.: Another Distributed Algorithm for Multiple Entries to a Critical Section, *Information Processing Letters*, Vol.41, No.1, pp.51–57 (1992).

(Received April 11, 1995)
(Accepted May 10, 1996)



Hirotsugu Kakugawa received the B.E. degree in electronics engineering in 1990 from Yamaguchi University, and the M.S. and D.E. degrees in information engineering in 1992, 1995 respectively from Hiroshima University. He is currently a Research Associate of Hiroshima University. His research interests include distributed algorithms, distributed systems, computer typography, and automata theory. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Masafumi Yamashita received the B.E. and M.E. degrees in information science in 1974, 1977, respectively from Kyoto University and the D.E. degree in information science in 1981 from Nagoya University. He worked from 1980 to 1985 as a Research Associate of Toyohashi University of Technology, and has been a member of the faculty of Engineering, Hiroshima University since 1985. He is currently a Professor at Department of Electrical Engineering, Hiroshima University. He has held several visiting appointments with Simon Fraser University. His research interests include distributed/parallel algorithms/systems.