Vol. 41 No. 2 Transactions of Information Processing Society of Japan Feb. 2000

Regular Paper

Distributed Resource Allocation among Overlapping Groups*

Zixue Cheng,† Yutaka Wada,† Yukiko Inoue,†
Yao Xue Zhang†† and Shoichi Noguchi†

The distributed resource allocation problem is a well known fundamental problem in distributed systems. Many solutions which avoid the deadlock and starvation have been developed. With the progress of computer networks, however, distributed cooperative group activities in a network environment have been increasing, so that several groups may compete for some resources in the network environment and *deadlock among groups* and *starvation of a group* may happen. Since previous allocation models are mainly for representation of competition for resources among processes, they cannot reflect clearly the competition for resources among groups of processes. Moreover, though the previous solutions to the distributed resource allocation problem can avoid the deadlock and starvation, they cannot deal with the deadlock among groups and starvation of a group. In this paper, we present a new model which described explicitly the competition for resources among process groups which may share common processes, and a definition of "Distributed Allocation of Resources to process Group" (DARG) under the model. A solution to *DARG* is also proposed by extending an acyclic graph approach to the dining philosopher problem. Our solution allocates resources to groups of processes with deadlock among groups and starvation of a group never happening. In addition, our algorithm guarantees that more than one group work mutual exclusively, if a common process belongs to these groups.

1. Introduction

A well known fundamental problem in distributed systems is the resource allocation problem, which could be briefly summarized as follows.

In a distributed system, there are a set of processes and a set of resources. Every process requires a subset of the resources and has to acquire all required resources for the process to perform its task. Every resource can be allocated to at most one process at a time. That is, the processes which require a resource have to access the resource mutual exclusively [1].

There are many variations of and extensions to the resource allocation problem. For example, Refs. 9) and 10) proposed the k-out of-M model which requires arbitrary k instances from M instances of a resource, and Refs. 5) and 9) gave the models which consider the types of resources.

When we consider the solution the resource allocation problem, the states in which no process can use resources and some process can never use resources have to be avoided. The former is called deadlock and the latter is called starvation. Both of them are important problems in development of distributed systems. Many distributed algorithms for avoiding them have been developed. Some examples of them are the methods which employ techniques such as time stamps [6),9),10)], acyclic directed graph [1)], and coterie [6)], etc.

On the other hand, with the progress of computer networks, many cooperative activities of groups exist in a network environment. If these groups' activities are performed around the same period of time, these groups may compete for resources of the network. In such a case, the deadlock among groups and/or starvation of a group may happen.

For example, there are 5 groups g_a, g_b, g_c, g_d, and g_e consisting of $\{p_1, p_2\}$, $\{p_4, p_5\}$, $\{p_3, p_6\}$, $\{p_7, p_8\}$, and $\{p_7, p_9\}$, respectively as shown in **Fig. 1**. A process may be an agent, a human, an object, or a UNIX process depending on different applications. The members of a group may be resident at different sites far from each other.

Processes in different groups may be geographically close to each other and compete for resources. In the example, p_1 and p_6 compete

† Department of Computer Software, University of Aizu
†† Department of Computer Science & Technology, TsingHua University

* This work is partially sponsored by a Research Grant from Fukushima Prefecture, National 973 Fundamental Research Program of China, and Chinese National Science Funds

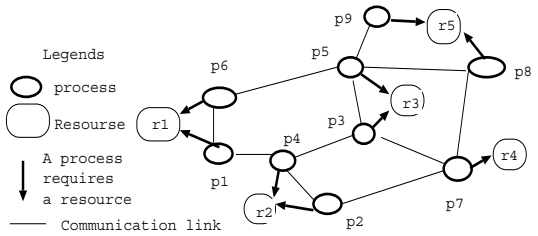

Fig. 1 An example of competition among groups.

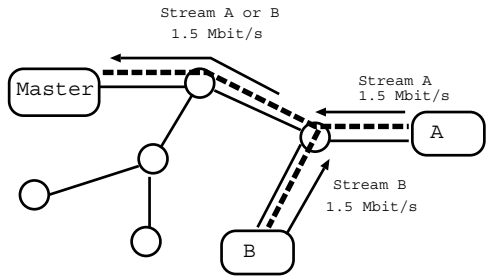

Fig. 2 An example of bandwidth sharing.

for r_1, p_2 and p_4 compete for r_2, p_3 and p_5 compete for r_3, and p_8 and p_9 compete for r_5.

To make the story more realistic, suppose every group wants to hold a group meeting around the same time, they compete for meeting rooms equipped with high quality presentation tools such as shared white board, screens and cameras for eye contact, and so on. Since the meeting rooms are limited, allocation of rooms to groups is important. Careless allocation, such as r_1 to p_1, r_2 to p_4, and r_3 to p_3, may lead to none of g_a, g_b, and g_c can hold its meeting, even some process may acquire the access privilege to some room, no group meeting can be held, since other members of the group cannot acquire meeting rooms.

Notice there is process p_7 belonging to both groups g_d and g_e. However, p_7 can participate in only one group's meeting at a time. We have to consider such a situation, to arrange a meeting time satisfying the condition, while we allocate the meeting rooms to groups. Careless arrangement may also lead to a deadlock among groups. For example, if allocating r_5 to p_9, but arranging p_7 to attend the meeting of g_d, both the meetings cannot be held.

Previous resource allocation model and its variations and extensions could not reflect the relations among groups explicitly, and did not give the definition of deadlock among groups and starvation of a group. Moreover, previous algorithms could only deal with the deadlock among processes and starvation of a process, but not deadlock among groups and starvation of a group. Our previous works [14)~16)] did not consider the situations where a process may belong to more than one group and participate in only one group's activity at a time. Thus the application possibility was restricted.

In this paper, we include a situation, in which some process may be shared by several groups, into our model which is an abstraction of many real applications. **Figure 2** shows an example of a game application, which is a simplified version of the example [17)]. The master delivers information related to the game contents to player A and B. Each player of A and B in turn sends update information to the master alternately, when taking an action. For sending update information from the player A (or player B) back to the master, both the player A and the master (or the player B and the master) have to reserve enough bandwidth, say 1.5 Mbits/s in the example. That means a group, consisting of the player A and the master (or the player B and the master), has to acquire all required resources which may be competed for with other applications.

The two groups has a common member, i.e., the master. To save the network resources, bandwidth sharing technique is employed [17)]. Two streams from the player A to the master and from the player B to the master share bandwidth. Support the bandwidth assigned to (reserved by) the master, player A, and player B is 1.5 Mbits/s, respectively. The master participate in the two groups alternately, i.e., it receives information from A and B alternately. It cannot receive the information from the two players at the same time, since that will require 2×1.5 Mbits/s by the master. In most of cases, A and B will play the game in turn, and reserving 2×1.5 Mbits/s for the master is a waste of resource.

Figure 3 illustrates another example regarding cooperative development of software systems, where four project teams T_a, T_b, T_c, and T_d exist.

Each team consists of some members. In a team, every member basically works individually, e.g., coding a part of a program. Members in the team work cooperatively in an asynchronous way. In other words, everyone may work on his/her own part in a different time. However, besides individual works, the members of a team have to discuss simultaneously

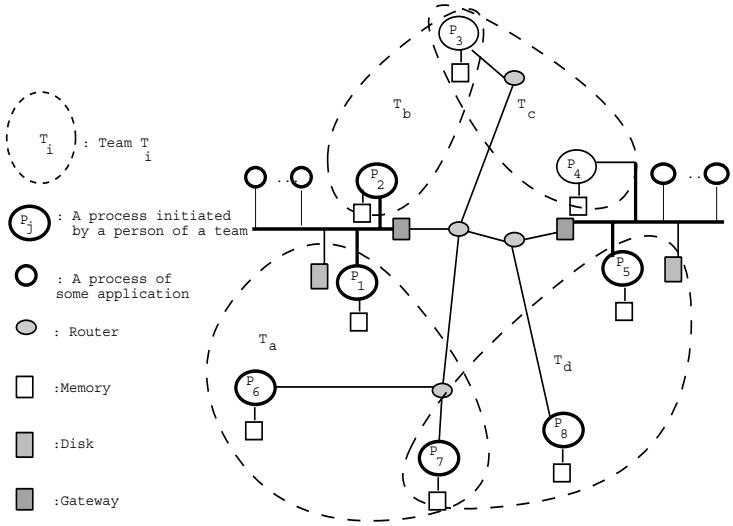

Fig. 3 An example of cooperative software development.

with one another by using a shared working space, once (or several times) a day. In order for a team to construct the shared working space, every member of the team has to acquire a certain amount of memory and network resource which may be competed for by other processes of other applications.

Because p_2 belonging to T_b is resident in a location close to p_1 of T_a, they may compete for network resource for sending out their data to other members of their teams, if the network resource is being used by many other applications and the network traffic load around the location is very high, it is possible that the two streams cannot send their data at the same time. Similarly p_4 and p_5 may compete for network resources.

Suppose a skillful person p_3 belongs to teams T_b and T_c, since p_3's specialty is needed by both teams. p_3 cannot attend the discussion of both teams at the same time, since p_3 wants to concentrate on one work at a time. The certain amount of memory assigned to p_3 is used for team T_b or T_c at a time. When p_3 finishes (suspends) the discussion with a team, the content of the memory will be written on a disk, so that the memory can be used for another team. Similar thing is true for another person p_7, who belongs to teams T_a and T_d.

Suppose the four groups have to discuss once (or several times) a day. There is a high possibility that those groups arrange the discussions around the same time. When we consider to allocate network resources to p_1, p_2, p_4 and p_5, we also have to take p_3 and p_7 into account, otherwise deadlock among groups may happen. For example, if we let p_2 use the network resource after p_1 finishes using, and p_5 use the network resource after p_4 finishes using, and we further assume that p_3 is arranged to attend the discussion of T_b first and p_7 is arranged to attend the discussion of T_d first, no group can have their discussion, since every group is lack of a member to participate or network resources for sending out data.

In order to solve the new problem, we firstly represent clearly the relation among groups of processes, and then define the deadlock among groups and starvation of a group explicitly. In addition, we propose a solution which guarantees that deadlock among groups and starvation of a group never happen, based on an extension of acyclic graph approach. Finally, the message complexity of the proposed algorithm is analyzed. For a group to acquire all required resources, $O(m + n^2)$ messages are necessary, where m is the number of processes which may compete for some resources with any process of the group, and n is the number of processes of the group.

Recently, mutual exclusion method considering the relations of inter-groups and intra-group attracts some researchers' attention, and an algorithm for resolving the resource competition by using K logical ring was proposed [11]. However, in the model, for every group, a resource

is competed for by all processes in the group, which cannot be applied to the situation where every process in a group has to be ensured to acquire its required resources at the same time, and do their cooperative group work together. So far, there is no such model as ours, which deals with that a set of processes in a group compete for resources with other groups, and acquire all required resources for them to do their cooperative group work.

The rest of the paper is organized as follows. In Section 2, our resource allocation model and the definition of DARG are given. In Section 3, we propose our new solution to DARG. Section 4 discusses the message complexity of the algorithm. A variation of the algorithm is presented in Section 5. Section 6 concludes the paper.

2. Model and Problem

Definition 1 (Resource Allocation Model) The resource allocation model is represented by a tuple (B, F). $B = (V, E)$ is a bipartite graph, where $V = \mathbf{P} \cup \mathbf{R}$ is a set of nodes consisting of a set of processes denoted with $\mathbf{P} = \{p_1, p_2, \cdots, p_i, \cdots p_n\}$ and a set of resources denoted with $\mathbf{R} = \{r_1, r_2, \cdots, r_j, \cdots, r_m\}$. E is a set of edges $e = (p_i, r_j)$ between p_i and r_j. F is a family of subsets of processes which belong to $\mathbf{P}$. That is $F = \{g_1, g_2, \cdots, g_k, \cdots, g_h\}$, where $g_k \subseteq \mathbf{P}$. g_k is called "process group" or "group".

If a process may require a resource, there is an edge between them in B. We also say the process is adjacent to the resource. Each process is adjacent to one or more resources. The set of resources, to which a process p_i is adjacent, is denoted with $R_i \subseteq \mathbf{R}$. The set of processes, which requires a resource r_j, is denoted with $P_j \subseteq \mathbf{P}$. If $R_{i1} \cap R_{i2} \neq \{\}$, two processes p_{i1} and p_{i2} are said to be in competition. In other words, there is a resource r_j, such that $p_{i1} \in P_j$ and $p_{i2} \in P_j$. The set of processes, the set of resources, the adjacent relation between processes and resources, and the set of groups don't change dynamically during an execution of our algorithm.

The following properties have to be guaranteed when allocating resources to processes of groups.

(1) Each process has to acquire all resources adjacent to it.

(2) A resource can be allocated to only one process at a time.

(3) In order for a group to perform their group cooperative work, the group has to acquire all resources adjacent to the group (all resources adjacent to any process in the group).

(4) Two or more groups cannot work at the same time, if they share any common process (i.e., the process belongs to these groups).

Remark 1 The conditions (1) and (3) could be relaxed to a more general model, since in some applications, that a part of processes acquire a part of required resources is enough for a group to perform its group work.

Definition 2 (Deadlock, Starvation, Group Deadlock, and Group Starvation)
Deadlock means that no process can acquire all resources adjacent to it.
Group Deadlock (*deadlock among groups*) means that there is no group such that every process of the group can acquire all resources adjacent to it.
Starvation means that there is some process that never acquire all resources adjacent to it at the same time.
Group Starvation (*starvation of a group*) means that there is some group such that some process of the group never acquire all resources adjacent to it at the same time.

Assumption 1 We assume that processes belonging to the same group do not compete for any resource. That is because if they compete for a resource, not all processes in the group can acquire their required resources at the same time. That means it is impossible to solve the problem with group starvation free.

However, the assumption could be removed under the relaxed condition mentioned in remark 1.

Definition 3 (Distributed Problem DARG) The problem DARG (Distributed Allocation of Resources to process Groups) is how to devise an algorithm which allocates the resources to the processes in the above model without group deadlock, group starvation, deadlock, and starvation under the above assumption.

Initially, each process knows its own identifier, the identifiers of groups to which the process belongs, identifiers of all processes in these groups, identifiers of the processes which compete for some resource(s) with it. We assume that a total order among group identifiers denoted with integers is predefined. Similarly, another total order among process identifiers is predefined also.

As output, each process in a group acquires

all required resources and begins to work by using these resources, with Deadlock, Starvation, Group Deadlock, and Group Starvation never happening.

Assumption 2 (Underlying Network)
In this paper, we consider that the resource allocation model will be implemented in a network environment. A process is mapped to a site of the network. Each pair of processes in the same group are connected with a communication channel. In addition to these channels, there is a communication channel between each pair of processes which are in competition.

Each process executes the same algorithm, which consists of sending massages on incident edges, waiting for incoming messages, and processing them. Messages are transmitted independently in both directions in a communication channel, and arrive after a finite but unpredictable delay, without errors and in FIFO order.

Each process has 3 states: *thinking*, *hungry*, and *acquired* similar to the 3 states of a philosopher in dining philosophers problem. In *thinking* state, the process does not require any resource. A process spontaneously makes transition from *thinking* to *hungry* state in finite time. "Spontaneously" means that the transition is not dependent on other processes, but the process itself. Generally, some input from a user or an application of the process initiates the transition. In *hungry* state, the process requires all resources adjacent to it. When all resources required by a process are acquired, its state is changed to *acquired*. Though a process in *acquired* state can access the resources, in order to use these resources for group work, the process may have to wait for other processes in the group, due to the properties (3) and (4) in Definition 1.

3. The Distributed Algorithm

3.1 Review of the Techniques of Previous Works

The traditional distributed resource allocation without considering competition among groups can be described as the distributed dining philosophers problem, illustrated by a connected undirected graph, in which a vertex (a process) represents a philosopher, and an edge represents a fork between the pair of processes that compete for a set of resources.

A famous solution for distributed dining philosophers problem is based on an acyclic directed graph technique [1),2)]. The technique points of the approach are summarized below. We assume that readers are familiar with the traditional dining philosopher problem. We use the terms such, thinking, hungry, eating, fork, etc. as in the problem without explanation.

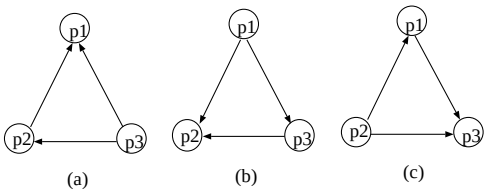

Fig. 4 An example of competition among three processes.

(1) An arc (p_i, p_j) oriented away from p_i to p_j means that p_j has the privilege to use the fork competed by the two processes.

(2) Initially, the direction of every arc is arranged such that the directed graph formed by the arcs are acyclic, so that there is at least a sink vertex (philosopher), which holds all privileges of forks incident to it and is able to acquire all required forks in order to eat.

(3) After eating, the philosopher reverses all incident arcs simultaneously. So that the directed graph is still acyclic and some other vertex becomes sink.

For example, 3 processes compete for 3 forks as shown in **Fig. 4** (a). Initially, p_1 is a sink, and eventually can acquire all required forks and use them. Then it reverses the arcs, see Fig. 4 (b), so that p_2 can acquire all required forks. The same will happen to p_3.

3.2 Some Technique Points of our Previous Solution

We extended the traditional resource allocation problem to the distributed resource allocation problem among process groups, where all processes of a group have to acquire all required resources in order to perform their cooperative works by using the resources, assuming there is no more than one group shares a common process [14),15)].

A solution for the extended problem was also based on the acyclic graph approach. However, the above technique points are not enough to solve the extended problem. For example, in **Fig. 5** (a), the graph is acyclic, but still a deadlock among groups happens, since there is no such a group that all processes in the group can acquire all required resources.

One reason of the group deadlock is because the arcs between g_a and g_b are inconsistent.

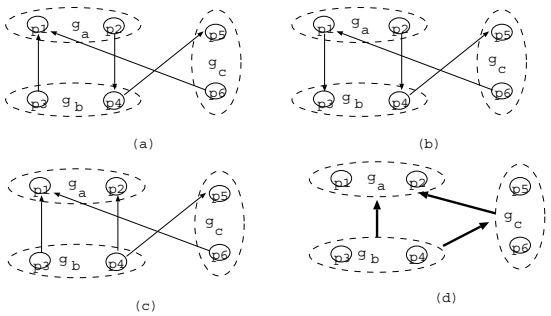

Fig. 5 Competition among three groups.

Namely, arc from p_3 to p_1 is oriented away from g_b to g_a, but arc from p_2 to p_4 is in the opposite direction. Therefore, g_a and g_b may never acquire all resources required. Even though the direction of the arc from p_3 to p_1 is reversed to remove the inconsistency, as shown in Fig. 5 (b), still group deadlock exists, since no group of g_a, g_b, and g_c can acquire all resources adjacent to the group. This is because for every group, some arc of a process belonging to the group is directed to a process of another group, which means the group doesn't hold all privileges. If reversing the arc from p_2 to p_4 in Fig. 5 (a), the deadlock is avoided, because all processes p_1 and p_2 can acquired all required resources (see Fig. 5 (c)).

Therefore in additional to above technique points, the following are required.

(4) All arcs between a pair of groups have to be oriented away from a group to another group.

(5) If all arcs between a pair of groups are considered as an meta-arc and a group as a meta-vertex, we obtain a meta-graph (see Fig. 5 (d)).

The meta-graph should be kept to be acyclic to guarantee group deadlock and group starvation.

3.3 Basic Ideas of the Solution in this Paper

In this paper, we consider *DARG* under a more general model, in which a process may belong to more than one group, compared to the model in Refs. 14) and 15). The above techniques (4) and (5) don't work in the generalized model. For example, in **Fig. 6**, if all arcs are oriented away from group g_c to g_a, it is not true for g_b and g_d. It is impossible for every pair of the groups to satisfy (4), whatever the initial directions of the arcs are set. The example tells us that we need more tricky and elegant control method for the generalized problem.

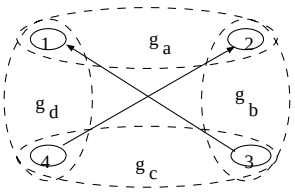

Fig. 6 An example of competition among overlapping groups.

In order to solve the problem, in the rest part of this subsection, we first introduce some new variables used by a process which belongs to more than one group. The values of these variables reflect which group has privilege to use the process and resource adjacent to the process. Based on the variables, we present 3 conditions which should be set initially and maintained during the execution of the algorithm, to guarantee group deadlock-free and group starvation-free. Also we outline how to maintain the conditions.

3.3.1 Variables

(1) Let $G_i \subseteq F$ be a set of groups, to each of which p_i belongs, and $|G_i| = q$. Every process p_i belonging to $q > 1$ groups holds a $q \times q$ array A_q. An element $x_{ig_k}^{ig_l}$ $(g_k \neq g_l)$ of A_q is a boolean variable, where $g_k, g_l, \in G_i$. $x_{ig_k}^{ig_l} = 1$ (hereafter we use 1 and 0 to represent the boolean values *true* and *false*) means that p_i would like to participate in the group work of g_k but not g_l. In other words, g_k holds the privilege to use p_i. $x_{ig_l}^{ig_k} = \neg x_{ig_k}^{ig_l}$ holds always. For the case of $g_k = g_l$, $x_{ig_k}^{ig_l} = null$.

In order for p_i to participate in activity of group g_k, $\forall g_l \in G_i, x_{ig_k}^{ig_l}$ must be equal to 1. After p_i finishes g_k's activity, $x_{ig_k}^{ig_l}$ is set to 0 and $x_{ig_l}^{ig_k}$ is set to 1.

Intuitively, a process belonging to $q > 1$ groups could be represented with q copies, each of which belongs to a different group, and there is an arc between every pair of the copies, as shown in **Fig. 7**. The variables described above could be considered as an implementation method of the arcs.

Remark 2 It seems to be redundant that i is used in both the subscript and superscript of $x_{ig_l}^{ig_k}$. However, it is for specifying variables in an uniform manner, which is convenient to specify the conditions of the algorithm. (See the latter part of the section.)

(2) In the solution for the dining philosophers problem [1),2)], an arc between a pair of pro-

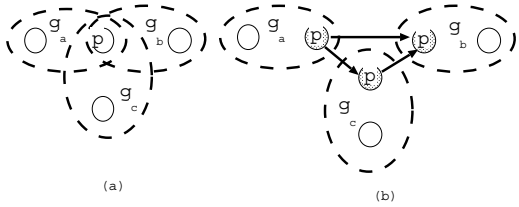

Fig. 7 Copies of a process shared by groups.

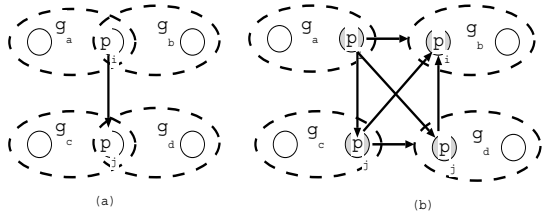

Fig. 8 Relations between copies of processes.

cesses (p_i, p_j) could be implemented by a pair of boolean variables x_i^j and x_j^i held by p_i and p_j, respectively, under the condition $x_i^j = \neg x_j^i$.

In this paper, if p_i and p_j belongs to q and r groups respectively, p_i holds a $q \times r$ array A_{qr} and p_j holds a $r \times q$ array A_{rq} of variables, for the edge (fork) between p_i and p_j. Element $x_{ig_k}^{jg_l}$ in A_{qr} is equal to $\neg x_{jg_l}^{ig_k}$ in A_{rq}.

Intuitively, p_i and p_j are split into q and r copies, respectively. These copies are connected by a directed complete bipartite graph. These variables are an implementation method of the directed arcs as shown in **Fig. 8**.

3.3.2 Conditions

In order to guarantee mutual exclusion, group deadlock and starvation freedom, the following three conditions have to be satisfied.

$$\forall i, j \in P, \quad g_k, g_l \in F \ s.\ t.\ g_k \neq g_l,$$
$$\left(x_{ig_k}^{jg_l} \wedge \neg x_{jg_l}^{ig_k}\right) \vee \left(\neg x_{ig_k}^{jg_l} \wedge x_{jg_l}^{ig_k}\right) = 1 \quad (1)$$

The condition is for the mutual exclusion property. That means, when $i \neq j$, no two processes use the same fork at the same time, and when $i = j$, no two groups sharing a common process perform their group tasks at the same time. Intuitively, the arc between two processes p_i and p_j is directed from p_i to p_j or vice versa. Also, the arc between two copies $p_i(g_k)$ and $p_i(g_l)$ of process $p_i = p_j$ is directed from $p_i(g_k)$ to $p_i(g_l)$ or vice versa.

$$\exists g_k, g_l \in F, \text{ s.t. } g_k \neq g_l,$$
$$\left(\bigwedge_{\forall i \in g_k,\ j \in g_l,\ \text{s.t. } x_{ig_k}^{jg_l} \text{ exists}} x_{ig_k}^{jg_l}\right) \bigvee$$
$$\left(\bigwedge_{\forall i \in g_k,\ j \in g_l,\ \text{s.t. } x_{ig_k}^{jg_l} \text{ exists}} \neg x_{ig_k}^{jg_l}\right) = 1 \quad (2)$$

The condition is for the mutual exclusion between two groups. All privileges should be held by one group to guarantee group starvation-freedom. Intuitively, all arcs between g_k and g_l should be directed from g_k to g_l or vice versa.

If processes $p_i \in g_k$ and $p_j \in g_l$ compete for a resource or g_k and g_l share a common process, we say group g_k and g_l are adjacent. A path among groups is defined as a sequence of distinct groups $g_1, g_2, \cdots, g_s$, such that g_{h+1} is adjacent to g_h, where $1 \leq h < s$. A circuit of groups is defined as a closed path such that all groups are distinct except $g_1 = g_s$.

For any circuit of groups, $g_1, g_2, \cdots, g_s$,

$$\left(\bigwedge_{\forall i_1, j_1, l_1 = g_2 \text{ or } g_{s-1}} x_{i_1 g_1}^{j_1 l_1}\right) \bigvee$$
$$\left(\bigwedge_{\forall i_2, j_2, l_2 = g_3 \text{ or } g_1} x_{i_2 g_2}^{j_2 l_2}\right) \bigvee \cdots \bigvee$$
$$\left(\bigwedge_{\forall i_{s-1}, j_{s-1}, l_{s-1} = g_s \text{ or } g_{s-2}} x_{i_{s-1} g_{s-1}}^{j_{s-1} l_{s-1}}\right) = 1 \quad (3)$$

The condition is for group deadlock-free. That means, for any circuit of groups at least there exists one group, whose variables related with the groups of the circuit are equal to 1. Intuitively, taking group as a node, the graph formed by the groups are acyclic. Thus there must be at least a group g_k' which is a sink in the graph, such that all its variables $x_{ig_k'}^{jg_l} = 1$.

3.3.3 Outline of the Methods for Maintaining the Conditions

Assuming that every group has a unique identifier represented by an integer, a simple method could be used to set the initial values of the variables such that all the above three conditions are satisfied.

For the pair of variables $x_{ig_k}^{ig_l}$ and $x_{ig_l}^{ig_k}$, held by process p_i, p_i sets $x_{ig_k}^{ig_l} = 1$ and $x_{ig_l}^{ig_k} = 0$, if $id(g_k) < id(g_l)$, where $id(g_k)$ and $id(g_l)$ denote the identifiers of g_k and g_l, respectively.

For the pair of variables $x_{ig_k}^{jg_l}$ and $x_{jg_l}^{ig_k}$ held by different processes p_i and p_j belonging to different groups, p_i and p_j send their group identifiers to each other, and then the one holds smaller

group identifier sets its corresponding variables to 1, another one sets its corresponding variables to 0.

The above three conditions are maintained during an execution of the algorithm (see the algorithm for detail). The basic points are as follows. After a group g_k such that all its variables $x_{ig_k}^{jg_l}$ are equal to 1 uses its resources, the privileges of using the resources should be transfered to adjacent groups. In other words, $x_{ig_k}^{jg_l}$ should be set to 0, and $x_{jg_l}^{ig_k}$ should be set to 1. Here $g_k \neq g_l$, i may (or may not) be equal to j. If $i = j$, $x_{ig_k}^{jg_l}$ and $x_{jg_l}^{ig_k}$ are held by the same process p_i. Thus they can be set by p_i. If $i \neq j$, $x_{ig_k}^{jg_l}$ and $x_{jg_l}^{ig_k}$ are held by p_i and p_j respectively.

Variable $x_{ig_k}^{jg_l}$ is set to 0 by p_i, and a $turn(g_k)$ message is sent to $p_j \in g_l$ by p_i. On receiving the message, p_j sets its variable $x_{jg_l}^{ig_k} = 1$. During the transmission of the message, variables $x_{ig_k}^{jg_l}$ and $x_{jg_l}^{ig_k}$ may be both equal to 0, but they will never be 1 at the same time, which guarantees the mutual exclusive access to resources.

After group g_k finishes using the resources, though it transfers the privileges to adjacent groups, it does not release the resources at once. It will release them only on request of other processes as in the solution for the distributed dining philosophers problem [2]. If other adjacent groups don't require the resources, i.e., the processes of these groups are in *thinking* state, the former group may use again the resources if it makes transition from *thinking* to *hungry* again.

To represent which one of p_i and p_j competing for a resource (fork) holds the fork, a variable y_i^j is employed. The fork competed for by p_i and p_j can be initially held by any one of p_i and p_j. We use a variable $y_i^j = 1$ to represent p_i holds the fork. Obviously, $y_j^i = \neg y_i^j$.

A group of processes will use the resources, only when all $x_{ig_k}^{jg_l} = 1$ and all its processes acquire all their required resources. To check if the condition is satisfied, communication among processes in the group is needed. *Inform* messages are employed for the purpose.

3.4 An Example

We show how our algorithm works for an example where 4 processes belonging to 4 groups compete for two resources as shown in **Fig. 9** (a).

We assume that the order on group identifiers is defined as $id(g_a) < id(g_b) < id(g_c) < id(g_d)$. For simplicity, we use a, b, c, and d to represent the identifiers of groups.

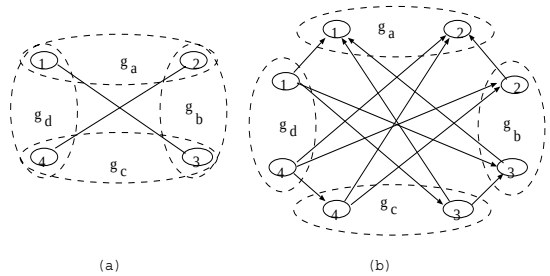

Fig. 9 An example of the algorithm.

Table 1 Initial values of the variables.

p_1		p_2	
g_a	g_d	g_a	g_b
$x_{1a}^{3b} = 1$	$x_{1d}^{1a} = 0$	$x_{2a}^{2b} = 1$	$x_{2b}^{2a} = 0$
$x_{1a}^{3c} = 1$	$x_{1d}^{3b} = 0$	$x_{2a}^{4c} = 1$	$x_{2b}^{4c} = 1$
$x_{1a}^{1d} = 1$	$x_{1d}^{3c} = 0$	$x_{2a}^{4d} = 1$	$x_{2b}^{4d} = 1$
p_3		p_4	
g_b	g_c	g_c	g_d
$x_{3b}^{1a} = 0$	$x_{3c}^{1a} = 0$	$x_{4c}^{2a} = 0$	$x_{4d}^{2a} = 0$
$x_{3b}^{3c} = 1$	$x_{3c}^{3b} = 0$	$x_{4c}^{2b} = 0$	$x_{4d}^{3b} = 0$
$x_{3b}^{1d} = 1$	$x_{3c}^{1d} = 1$	$x_{4c}^{4d} = 1$	$x_{4d}^{4c} = 0$

Every process holds variables $x_{ig_k}^{jg_l}$ as shown in **Table 1**, where the identifiers of processes $i, j \in \{1, 2, 3, 4\}$ and the identifiers of groups $g_k, g_l \in \{a, b, c, d\}$. These variables are initiated to 1 or 0, with the method mentioned in Section 3.1. The initial state could be represented by a directed graph shown in Fig.9 (b).

Whatever the value y_i^j is, i.e., whichever process of p_i and p_j holds the fork, since all variables of p_1 and p_2 in group g_a are equal to 1, i.e. $x_{1a}^{1d} = 1$, $x_{2a}^{2b} = 1$, $x_{1a}^{3b} = 1$, $x_{1a}^{3c} = 1$, $x_{2a}^{4c} = 1$, and $x_{2a}^{4d} = 1$, p_1 and p_2 will not send forks on receiving *request* messages, until they acquire all required resources, and finish performing their group work using the resources. In other words, p_1 and p_2 will eventually acquire all required resources, send an *inform* message to each other, and access the resources. After that, p_1 and p_2 set $x_{ia}^{ig_l} = 0$, $x_{ia}^{jg_l} = 0$, and $x_{ig_l}^{ia} = 1$. Also, p_1 and p_2 send a $turn(a)$ message to p_3 and p_4, respectively.

The variables $x_{jg_l}^{ia}$ will be set to 1 by p_3 and p_4, on receiving $turn(a)$. See **Table 2** for the change of the values of the variables.

From Table 2, all variables of p_2 and p_3 in group g_b are equal to 1. Therefore, p_2 and p_3 will eventually acquire all required resources, send an *inform* message to each other, and access the resources. After that, p_2 and p_3 set $x_{ib}^{ig_l} = 0$, $x_{ib}^{jg_l} = 0$, and $x_{ig_l}^{ib} = 1$. Also, p_2 and

Table 2 Values of the variables after g_a is executed.

p_1		p_2	
g_a	g_d	g_a	g_b
$x_{1a}^{3b}=0$	$x_{1d}^{1a}=1$	$x_{2a}^{2b}=0$	$x_{2b}^{2a}=1$
$x_{1a}^{3c}=0$	$x_{1d}^{3b}=0$	$x_{2a}^{4c}=0$	$x_{2b}^{4c}=1$
$x_{1a}^{1d}=0$	$x_{1d}^{3c}=0$	$x_{2a}^{4d}=0$	$x_{2b}^{4d}=1$
p_3		p_4	
g_b	g_c	g_c	g_d
$x_{3b}^{1a}=1$	$x_{3c}^{1a}=1$	$x_{4c}^{2a}=1$	$x_{4d}^{2a}=1$
$x_{3b}^{3c}=1$	$x_{3c}^{3b}=0$	$x_{4c}^{2b}=0$	$x_{4d}^{2b}=0$
$x_{3b}^{1d}=1$	$x_{3c}^{1d}=1$	$x_{4c}^{4d}=1$	$x_{4d}^{4c}=0$

Table 3 Values of the variables after g_b is executed.

p_1		p_2	
g_a	g_d	g_a	g_b
$x_{1a}^{3b}=1$	$x_{1d}^{1a}=1$	$x_{2a}^{2b}=1$	$x_{2b}^{2a}=0$
$x_{1a}^{3c}=0$	$x_{1d}^{3b}=1$	$x_{2a}^{4c}=0$	$x_{2b}^{4c}=0$
$x_{1a}^{1d}=0$	$x_{1d}^{3c}=0$	$x_{2a}^{4d}=0$	$x_{2b}^{4d}=0$
p_3		p_4	
g_b	g_c	g_c	g_d
$x_{3b}^{1a}=0$	$x_{3c}^{1a}=1$	$x_{4c}^{2a}=1$	$x_{4d}^{2a}=1$
$x_{3b}^{3c}=0$	$x_{3c}^{3b}=1$	$x_{4c}^{2b}=1$	$x_{4d}^{2b}=1$
$x_{3b}^{1d}=0$	$x_{3c}^{1d}=1$	$x_{4c}^{4d}=1$	$x_{4d}^{4c}=0$

Table 4 Values of the variables after g_c is executed.

p_1		p_2	
g_a	g_d	g_a	g_b
$x_{1a}^{3b}=1$	$x_{1d}^{1a}=1$	$x_{2a}^{2b}=1$	$x_{2b}^{2a}=0$
$x_{1a}^{3c}=1$	$x_{1d}^{3b}=1$	$x_{2a}^{4c}=1$	$x_{2b}^{4c}=1$
$x_{1a}^{1d}=0$	$x_{1d}^{3c}=1$	$x_{2a}^{4d}=0$	$x_{2b}^{4d}=0$
p_3		p_4	
g_b	g_c	g_c	g_d
$x_{3b}^{1a}=0$	$x_{3c}^{1a}=0$	$x_{4c}^{2a}=0$	$x_{4d}^{2a}=1$
$x_{3b}^{3c}=1$	$x_{3c}^{3b}=0$	$x_{4c}^{2b}=0$	$x_{4d}^{2b}=1$
$x_{3b}^{1d}=0$	$x_{3c}^{1d}=0$	$x_{4c}^{4d}=0$	$x_{4d}^{4c}=1$

Table 5 Values of the variables after g_d is executed.

p_1		p_2	
g_a	g_d	g_a	g_b
$x_{1a}^{3b}=1$	$x_{1d}^{1a}=0$	$x_{2a}^{2b}=1$	$x_{2b}^{2a}=0$
$x_{1a}^{3c}=1$	$x_{1d}^{3b}=0$	$x_{2a}^{4c}=1$	$x_{2b}^{4c}=1$
$x_{1a}^{1d}=1$	$x_{1d}^{3c}=0$	$x_{2a}^{4d}=1$	$x_{2b}^{4d}=1$
p_3		p_4	
g_b	g_c	g_c	g_d
$x_{3b}^{1a}=0$	$x_{3c}^{1a}=0$	$x_{4c}^{2a}=0$	$x_{4d}^{2a}=0$
$x_{3b}^{3c}=1$	$x_{3c}^{3b}=0$	$x_{4c}^{2b}=0$	$x_{4d}^{2b}=0$
$x_{3b}^{1d}=1$	$x_{3c}^{1d}=1$	$x_{4c}^{4d}=1$	$x_{4d}^{4c}=0$

p_3 send a $turn(b)$ message to p_4 and p_1, respectively.

The further change of values of variable in each process during the execution is shown in **Tables 3**, **4**, and **5**. Readers can trace in the same way as described above.

Notice the Table 5 is same as the Table 1, which means the state of the system returns to the initial one. So the above execution steps will be repeated.

4. Correctness and Complexity of the Algorithm

4.1 Correctness

Theorem 1 Mutual exclusive access to every resource is preserved.

PROOF. When a resource is competed for by a pair of processes p_i and p_j, a pair of variables y_i^j and y_j^i are used by p_i and p_j respectively. Initially, only one of the two variables is set to *true*. Another is set to *false*. The process with its variable being *false* cannot access the resource. Without loss of generality, let's assume y_i^j = **true** and y_j^i= **false** . Then, p_j with its variable being *false* cannot access the resource and will send a *request* message to p_i. On receiving the *request* message, p_i will set y_i^j to **false** and send a *fork* message to p_j immediately, or postpone the setting and sending *fork* until it finishes using the resource. After receiving the *fork* message, p_j sets y_j^i to *true*. Before receiving the *fork* message, both of the variables may be *false*. But both variables never become *true* at the same time. Therefore, mutual exclusive access to the resource is preserved, since a process can access the resource, only when its variable becomes *true*. □

Lemma 1 Condition (3) in Section 3.3.2 is preserved by the algorithm.

PROOF. A unique group identifier is assigned to every group, and a total order is defined over the identifiers of all groups. The variables $x_{ig_k}^{jg_l}$ and $x_{jg_l}^{ig_k}$ are initialized according to the identifiers, such that the group with smaller identifier holds the privilege. Therefore, for every circuit, Condition (3) is satisfied. Consequently, there is at least a group $g_{k'}$ such that all variables $x_{ig_{k'}}^{jg_l}$ are equal to 1. The one with the smallest group identifier of all groups is such a group.

In addition, when $g_{k'}$ finishes its task and variables $x_{ig_{k'}}^{jg_l}$ and variables $x_{jg_l}^{ig_{k'}}$ are changed to 0 and to 1 respectively, Condition (3) are still preserved. This can be shown by the following case analysis.

(1) If in a circuit of groups g_1, g_2,$\cdots$, $g_{k'}$, $\cdots$, g_k, $\cdots$, g_{s-1}, $g_s(=g_1)$, there is another g_k besides $g_{k'}$, such that

$$\left(\bigwedge_{\forall i_k, j_k, l_k = g_{k+1} \text{ or } g_{k-1}} x_{i_k g_k}^{j_k l_k} \right) = 1,$$

Condition (3) still holds, even all $x_{ig_{k'}}^{jg_l}$ are set to 0.

(2) Suppose that in a circuit of groups $g_1, g_2, \cdots, g_{k'}, \cdots, g_{s-1}, g_s(= g_1)$ there is only a group $g_{k'}$ such that all variables of $g_{k'}$ are equal to 1, then variables $x_{ig_{k'+1}}^{jg_{k'}} = 0$ and $x_{ig_{k'-1}}^{jg_{k'}} = 0$. However, at least for one group of $g_{k'+1}$ and $g_{k'-1}$ adjacent to $g_{k'}$, variables $x_{ig_{k'+1}}^{jg_{k'+2}}$ or $x_{ig_{k'-1}}^{jg_{k'-2}}$ should be 1, otherwise, there will be another g_k as in Case (1) since the circuit is acyclic originally. When variables $x_{ig_{k'}}^{jg_l}$ are changed to 0, variables $x_{ig_{k'+1}}^{jg_{k'}}$ and $x_{ig_{k'-1}}^{jg_{k'}}$ will be changed to 1. Thus,

$$\left(\bigwedge_{\forall i_k, j_k, l_k = g_{k'} \text{ or } g_{k'+2}} x_{i_k g_{k'+1}}^{j_k l_k} \right) = 1 \text{ or}$$

$$\left(\bigwedge_{\forall i_k, j_k, l_k = g_{k'} \text{ or } g_{k'-2}} x_{i_k g_{k'-1}}^{j_k l_k} \right) = 1.$$

That means Condition (3) is preserved. □

Lemma 2 A group g_k such that all variables $x_{ig_k}^{jg_l} = 1$ will eventually execute its task.

Proof. In the case $i = j$, $x_{ig_k}^{jg_l} = 1$ means process p_i will participate in g_k rather than g_l. In the case $i \neq j$, $x_{ig_k}^{jg_l} = 1$ means process p_i holds the privilege to use the resource competed for by p_i and p_j.

If p_i is hungry and has held the resource, i.e., $y_i^j = 1$, it will not release the resource, until it finishes using the resource, even on receiving a *request* from p_j, since the condition "**not** $hungry_i$ **or not** $\exists g_k \in G_i, \forall g_l \in G_j$, s.t. $\bigwedge x_{ig_k}^{jg_l}$" is not satisfied, i.e., $hungry_i = 1$ and $\exists g_k \in G_i, \forall g_l \in G_j$, s.t. $\bigwedge x_{ig_k}^{jg_l} = 1$ (see the part of algorithm for receipt of *request* messages).

If p_j holds the resource, i.e., $y_j^i = 1$, on receiving a *request* message from p_i, p_j will response the *request* and send $fork$ immediately, since the condition "**not** $\exists g_l \in G_j, \forall g_k \in G_i$, s.t. $\bigwedge x_{jg_l}^{ig_k}$ " of the algorithm for p_j is satisfied.

Therefore, g_k will acquire all required resources and executed its task. □

Theorem 2 Group deadlock never happen.

Proof. By Lemma 1, during an execution of the algorithm, there is always at least a group g_k such that all its variables $x_{ig_k}^{jg_l}$ are equal to 1. By Lemma 2, such a group will eventually execute its task. Therefore, the group deadlock never happen. □

Notation 1 (Depth of a Group)
We call a group *sink group* if values of all its variables are equal to 1 . The depth of a sink group is defined as 0. Due to Condition (3), for every group, say g_1, there is a finite length directed path $g_1, g_2, \cdots, g_k \cdots, g_s$ from g_1 to a sink group g_s, such that $x_{ig_k}^{jg_{k+1}} = 0$ $(1 \leq k \leq s-1)$. The length of the longest path of a group (out of all paths from the group) is called the depth of the group. □

Lemma 3 If $x_{ig_k}^{jg_l} = 0$, the depth of g_k is greater than that of g_l.

Proof. Suppose the longest path of g_l is $g_l, g_{l+1}, \cdots, g_k \cdots, g_s$ from g_l to a *sink group* g_s and the depth of group g_l is d_l. Because of $x_{ig_k}^{jg_l} = 0$, there is a path $g_k, g_l, g_{l+1}, \cdots, g_k \cdots, g_s$ from g_k to g_s and the length of the path is $d_l + 1$. So the depth of g_k is not less than $d_l + 1$, due to the the definition of depth of a group. □

Theorem 3 Group starvation will never happen.

Proof. From the definition of depth of a group and Lemma 2, a group whose depth is 0 will eventually acquire all required resources and execute its task.

In the following, we show inductively that the depth of every group will eventually becomes 0.

(1) The depth of a group g_k, whose current depth is 1, will eventually change to 0.

Variables $x_{ig_k}^{jg_l}$ of g_k, whose current depth is 1, is equal to 0, only if the depth of g_l is 0. Otherwise, $x_{ig_k}^{jg_{l'}} = 1$.

Since group g_l whose current depth is 0 will eventually acquire all resources, execute its task, and send a *turn* message to g_k, variables $x_{ig_k}^{jg_l}$ will eventually be set to 1, i.e., all g_k's variables becomes 1. That means the depth of g_k becomes 0.

(2) Suppose the depth of group, whose current depth is less than k, will eventually become 0, we show the depth of a group whose current depth is k will eventually become 0.

Variables $x_{ig_k}^{jg_l}$ of a group g_k, whose current depth of is k, is equal to 0, only if the depth of g_l is less than k. Otherwise, $x_{ig_k}^{jg_{l'}} = 1$, (where the depth of $g_{l'}$ is not less than k).

By the assumption of the induction, every group g_l whose current depth is less than k will eventually acquire all resources, execute its task, variables $x_{ig_k}^{jg_l}$ will eventually be set to 1, i.e., all g_k's variables becomes 1. That means

the depth of g_k becomes 0. □

4.2 Message Complexity

To compute the complexity of the algorithm, we denote the number of processes which may compete for a resource with p_i by ρ_i.

$$\rho_i = \left| \bigcup_{r_j \in R_i} P_j \right| - 1 = |Neig_i|$$

Namely, ρ_i is the number of processes which are adjacent to any of resources required by p_i.

Let $n = |g_k|$, and m be the number of processes which may compete for any resource with any process in group g_k. That is,

$$m = \sum_{p_i \in g_k} \rho_i.$$

The maximum number of messages (the worst case complexity) which are needed in our algorithm for a group g_k to work once is computed below.

For a group g_k to work once, a *request* message is sent by every $p_i \in g_k$ to every $p_j \in Neig_i$ which may compete for some resources with p_i, a *fork* message is sent back by $p_j \in Neig_i$ to $p_i \in g_k$, an *inform* message is sent by every $p_i \in g_k$ to every $p_h \in g_k$ except p_i itself, and a *turn* message is sent by every $p_i \in g_k$ to every $p_j \in Neig_i$.

Thus, the total messages is:

$$\sum_{p_i \in g_k} \rho_i + \sum_{p_i \in g_k} \rho_i + \sum_{p_i \in g_k} (n-1) + \sum_{p_i \in g_k} \rho_i = 3m + n(n-1).$$

The message complexity is $O(m + n^2)$.

5. A Variation

In the algorithm presented in Section 3, we assume that processes become hungry, independently. That is, a process is independent of other processes to change its state from *thinking* to *hungry*. However, in some applications, a group of processes become *hungry* in the same time, in order to carry a group work. Therefore, it is necessary to deal with the situation of *group hungry*. For example, let's suppose that group g_a and g_c are planing to have a group meeting, but neither g_b nor g_d do, in Fig. 6. Though all processes of p_1, p_2, p_3, and p_4 become hungry and acquire all required resources, g_b and g_d should not work, because they are not *hungry* as a group.

A variation of algorithm is given below to deal with such kind of applications. In addition to the variable $hungry_i$, for every $g_k \in G_i$, p_i holds a variable $hungry_i^k$ which will be set to *true* by the application program or the user of p_i. After a hungry process p_i acquires all required resources, it will send an *inform* message to all processes in one hungry group $g_a \in G_i$ which has the most privileges in all hungry groups, where we say g_k has more privileges than g_l, if $x_{ig_k}^{ig_l} = 1$. If group $g_b \in G_i$, having more privileges than g_a, becomes *hungry*, before p_i receives an *inform* message from every process in g_a, p_i sends a *cancel* message to each of g_a. A process p_j in g_a will send a *delete* message to every process in g_a to let them delete the *inform* from p_i, if p_j receives *cancel*, before it issues an *inform* to each of g_a. Process p_i will send an *inform* to every process in g_b, on receiving a *delete* message from p_j. A process in g_a will execute g_a's task, if it receives an *inform* from every process in g_a, and discards *cancel* messages received.

If any process receives a *delete* message issued by another process which will issue an *inform* message to every process in the group after a *delete* message, every process in the group will receive the *delect* message before the *inform* message, due to the FIFO property of a channel. Thus none will execute the group's task. If a process receives an *inform* from every other process, it means every other process has sent the *inform* before receiving a *cancel*. Thus, none of them will send a *delete* message. Consequently, every process in the group will receive an *inform* messages from all others, and executes the group's task. Therefore consistency of the group is ensured.

6. Conclusion

In this paper, we have presented a generalization of the resource allocation problem and a solution to the generalized problem.

Recently, opportunities for groups of people or agents to work together on a computer network are increasing. The demand for applications that support such works is growing now and in the near future. The problem of distributed resource allocation among process groups can be considered as a building block for such applications.

Comparing with our previous work, we removed the restriction that no more than one group may share a common process, so that the new solution could be used by more applications.

Some future works are summarized as fol-

lows:

1. For a given set of processes, how to partition the processes into groups is an interesting problem and needs to be further studied.
2. The time complexity of our solution needs to be analyzed.

Acknowledgments The authors would like to thank Dr. Yoshifumi Manabe and the anonymous referees for their critical comments which improve the paper a lot. The authors' thanks also go to the staff and students at the Computer Network Laboratory, Department of Computer Software, University of Aizu, for their valuable discussion and suggestions on the work.

References

1) Chandy, K.M. and Misra, J.: The Drinking Philosophers Problem, *ACM Trans. Prog. Lang. Syst.*, Vol.6, No.4, pp.632–646 (1984).
2) Barbosa, V.C.: *An Introduction to Distributed Algorithms*, pp.233–240, MIT Press (1996).
3) Lynch, N.A.: *Distributed Algorithms*, Morgan Kaufmann Publishers (1996).
4) Rhee, I: A fast Distributed Modular Algorithm for Resource Allocation, *Proc. 15th International Conference on Distributed Computing Systems*, pp.161–168 (1995).
5) Ginat, D., Shankar, A.U. and Agrawala, A.K.: An Efficient Solution to the Drinking Philosophers Problem and its Extensions, *Proc. 3rd International Workshop on Distributed Algorithms*, *LNCS*, Vol.392, pp.83–93 (1989).
6) Kakugawa, H. and Yamashita, M.: Local Coteries and a Distributed Resource Allocation Algorithm, *Trans. IPS Japan*, Vol.37, No.8, pp.1487–1496 (1996).
7) Lamport, L.: Time, Clock, and Ordering of Events in a distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
8) Maekawa, M.: A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, *ACM Trans. Comput. Syst.*, Vol.3, No.2, pp.145–159 (1985).
9) Raynal, M.: A Distributed Solution to the k-out of-M Resources Allocation Problem, *LNCS*, Vol.497, pp.599–609, Springer-Verlag (1991).
10) Baldoni, R.: An $O(N^{M/M+1})$ distributed algorithm for the k-out of-M resources allocation problem, *Proc. 14th ICDCS*, pp.81–88 (1994).
11) Khiat, A. and Naïmi, M.: Distributed Mutual Exclusion in K-Groups Based on K-Logical Rings, *Proc. 11th Annual International Symposium on High Performance Computing Systems* (*HPCS'97*), pp.269–282 (1997).
12) Bar-Ilan, J. and Peleg, D.: Distributed Resource Allocation Algorithms, *LNCS*, Vol.647, pp.277–291 (1992).
13) Huang, T. and Cheng, Z.: A Distributed Algorithm for Optimal Allocation of Resources, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'97*), pp.880–884 (1997).
14) Cheng, Z., Huang, T. and Shiratori, N.: A Distributed Algorithm for Resource Allocation among Process Group, *Proc. 9th International Conference on Information Networking*, pp.443–448 (1994).
15) Wada, Y., Cheng, Z. and Huang, T.: A Distributed Algorithm for Allocation of Resources to Process Groups with Acyclic Graphs, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA'97*), pp.801–805 (1997).
16) Wada, Y., Cheng, Z. and Noguchi, S.: Distributed Algorithms for Allocation of Resources to Process Groups and Their Complexity, *IEICE Trans. Information and Systems*, Vol.J81-D-I, No.6, pp.651–665 (1998).
17) Delgrossi, L.: *Design of Reservation Protocols for Multimedia Communication*, p.119, Kluwer Academic Publishers (1996).

Appendix

The formal description of the algorithm

Every process p_i holds the following variables.

- A boolean variable $x_{ig_k}^{ig_l}$ for each $g_k, g_l \in G_i$ $(g_k \neq g_l)$ is an element of the array $A_{|G_i|}$, where $G_i \subseteq G$ is the set of groups to which p_i belongs.
- A boolean variable $x_{ig_k}^{jg_l}$ for each $g_k \in G_i$, $p_j \in Neig_i$, $g_l \in G_j$, where $Neig_i$ is the set of processes, each of which competes for some resource with p_i.
- A boolean variable y_i^j for each $p_j \in Neig_i$ to represent whether p_i holds the fork competed for by p_i and p_j.
- A boolean variable z_i^j for each $p_j \in Neig_i$ to represent whether p_i has postponed to response to a p_j's request for the fork competed for by p_i and p_j. Initially $z_i^j =$ **false**
- A boolean variable $hungry_i$ is to represent whether p_i requires resources or not. Initially, $hungry_i =$ **false**;
- A boolean variable $informed_{ig_k}^h$ for each $p_h \in g_k$ *s.t.* $g_k \in G_i$ to represent whether p_i has received an *inform* message from p_h. Initially, $informed_{ig_k}^h =$ **false**;

- A boolean variable $inform_{ig_k}$ to represent whether p_i has sent *inform* messages to all processes in g_k. Initially, $inform_{ig_k}$ = **false**;

The initial values of variables $x_{ig_k}^{ig_l}$, $x_{ig_k}^{jg_l}$, and y_i^j are described in Section 3.1.
The types of messages used in the algorithm are as follows:

- *request*: sent from p_i to p_j, which compete for some resources with p_i, to ask for the *fork*.
- *fork*: sent from p_i to p_j, which compete for some resources with p_i, on p_j's request.
- *turn*: sent from p_i to p_j, which compete for some resources with p_i, right after p_i finishes using the resources.
- *inform*: sent from a process $p_i \in g_h$ to every $p_j \in g_h$ to let them know all resources required by p_i have been acquired.

Every process p_i executes the following algorithm.

When p_i makes transition from *thinking* to *hungry*
begin
$hungry_i$:= **true** ;
if y_i^j = **true** for all $p_j \in Neig_i$ **then**
if $\exists g_k \in G_i, \forall p_h \in g_k$, $informed_{ig_k}^h$ = **true**
then
Access()
else if $\exists g_k$,
$\forall p_j \in Neig_i, \forall g_l \in G_j, x_{ig_k}^{jg_l}$ = **true**
begin
send *inform* to all $p_h \in g_k$;
$inform_{ig_k}$:= **true**
end
else send *request* to every $p_j \in Neig_i$, such that y_i^j=**false**
end .

On receiving a *request* message from $p_j \in Neig_i$,
begin
if not $hungry_i$
or not $\exists g_k \in G_i, \forall g_l \in G_j$, s.t. $\bigwedge x_{ig_k}^{jg_l}$ = **true**
then
begin
y_i^j := **false** ;
send *fork* to p_j;
if $hungry_i$ **then**
send *request* to p_j
end
else
z_i^j := **true**
end .

On receiving a *fork* message from $p_j \in Neig_i$,
begin
y_i^j:= **true** ;
if $y_i^{j'}$ = **true** for all $p_{j'} \in Neig_i$ **then**
if $\exists g_k \in G_i, \forall p_h \in g_k$, $informed_{ig_k}^h$ = **true**
then
Access()
else if
$\exists g_k, \forall p_j \in Neig_i, \forall g_l \in G_j$, $x_{ig_k}^{jg_l}$ = **true**
begin
send *inform* to all $p_h \in g_k$;
$inform_{ig_k}$:= **true**
end
end .

On receiving an *inform* message from $p_j \in g_k \in G_i$
begin
$informed_{ig_k}^h$ = **true**;
if ($\exists g_k \in G_i, \forall p_h \in g_k$ $informed_{ig_k}^h$ = **true**
and $\forall p_j \in Neig_i$, y_i^j = **true**) **then**
Access()
end .

On receiving a $turn(g_l)$ message from $p_j \in Neig_i$,
begin
$\forall g_k \in G_i, x_{ig_k}^{jg_l}$:= **true** ;
if ($\exists g_k, \forall p_j \in Neig_i, \forall g_l \in G_j$, $x_{ig_k}^{jg_l}$ = **true**
and $\forall p_j \in Neig_i$, y_i^j = **true**) **then**
begin
send *inform* to all $p_h \in g_k$;
$inform_{ig_k}$:= **true**
end
end .

Access(){
repeat
begin
if ($inform_{ig_k}$ = **false**) **then**
begin
send *inform* to all $p_h \in g_k$;
$inform_{ig_k}$:= **true**
end
Access shared resources;
for all $g_l \in G_i$ s.t. $g_l \neq g_k$ **do**
$x_{ig_k}^{ig_l}$:= **false**; $x_{ig_l}^{ig_k}$:= **true**;
for all $p_k \in Neig_i$, $g_l \in G_k$ **do**
if $x_{ig_k}^{kg_l}$:= **true then**
$x_{ig_k}^{kg_l}$:= **false**;
for all $p_k \in Neig_i$, **send** $turn(g_k)$ to p_k;

```
      ∀p_h ∈ g_k, informed^h_{ig_k} := false
    end
  until ∀g'_k ∈ G_i, ∃p_h ∈ g'_k, informed^h_{ig'_k} = false
  hungry_i := false ;
  if z^j_i = true then
    begin
      z^j_i := false ;
      y^j_i := false ;
      send fork to p_j
    end
}
```

(Received November 17, 1998)
(Accepted November 4, 1999)

Zixue Cheng received the M.E. and Ph.D. degrees from Tohoku University in 1990 and 1993, respectively. He was an assistant professor from 1993 to 1999 and has been an associate professor since April, 1999, at the Department of Computer Software, University of Aizu. Currently he is working on distributed algorithms, network agents, and distance education. Dr. Cheng is a member of IEEE, ACM, and IEICE.

Yutaka Wada received B.S. and M.S. degrees in computer science and engineering from the University of Aizu. He is currently a Ph.D. candidate in graduate school of computer science and engineering, the University of Aizu. His primary research interests are distributed algorithms, distributed applications, and computer networks. He is a student member of the Information Processing Society of Japan (IPSJ).

Yukiko Inoue joined the University of Aizu in 1994 and received the B.S. degree in 1998. She was working in distributed algorithms.

Yao Xue Zhang was born in Hunan, China, on Jan. 5, 1956. He received the B.S. degree in electronic communication from Xidian University, Xian, China, in 1982. In 1986 and 1989, he received the M.S. degree and the Ph.D. degree in computer science from Tohoku University, Japan, respectively. He joined Tsinghua University in 1990 and has been a full professor since 1993, at the department of Computer Science, Tsinghua University, China. He was a visiting scientist in Laboritory of Computer Science MIT, U.S.A, in 1995, and a visiting professor in University of Aizu, Japan, in 1998, respectively. He is currently leading a research group to study network architecture including management and control method of Quality of Services provided by networks, protocol specification, synthesis, verification and implementation method, network interconnection including definitions of interconnection protocol, routing algorithm, design and implementation of routers for network interconnection and protocol conversion method, information system integration, and network application software.

Shoichi Noguchi was born in Tokyo on March 5th, 1930, and received his B.E., M.E., and D.E. degrees in Electrical Communication Engineering from Tohoku University, Sendai. He joined the Research Institute of Electrical Communication at Tohoku University in 1960, and was a professor at the same University from 1971 to 1993. Dr. Noguchi had been Director of the Computer Certer, Tohoku University, from 1984 to 1990 and Drector of Research Certer for Applied Information Science, Tohoku University, from 1990 to 1993. He had been a professor of Faculty of Engineering, Nihon University, from 1993 to 1997. Dr. Noguchi was the President of IPSJ (Information Processing Society of Japan) from 1995 to 1997. He has been the President of the University of Aizu since April 1997. His main area of interests is information science theory and compuer network fundamentals. He has also been active in the area of parallel processing, computer network architecture, and knowledge engineering fundamentals.