

# A modular algorithm for resource allocation

Injong Rhee

Department of Computer Science, North Carolina State University, 446 EGRC, Campus Box 7534, Raleigh, NC 27695-7534, USA  
(e-mail: rhee@csc.ncsu.edu)

Received: May 1997 / Accepted: May 1998

**Summary.** This paper concerns resource allocation in distributed message passing systems, i.e., the scheduling of accesses to exclusive system resources shared among concurrent processes. An efficient modular resource allocation algorithm is presented that uses any arbitrary resource allocation algorithm as a subroutine. It improves the performance of the subroutine by letting each process wait only for its currently conflicting processes, and therefore, allows more concurrency. For appropriate choices of the subroutine, we obtain resource allocation algorithms with the minimum worst case response times. Simulation studies were conducted which also indicate that on average, the obtained algorithms perform faster and require a smaller number of messages than other previously known algorithms, especially when resource contention among processes is high and the average time that a process remains in the critical region is large.

**Key words:** Dining philosophers – Resource allocation – Modular construction – Concurrency – Message passing distributed systems

## 1 Introduction

Distributed systems commonly include exclusive resources, such as files and distributed objects, that must be managed so that no two processes access the resources at the same time, while avoiding starved or deadlocked processes. The scheduling of processes with various resource requirements in this type of system is generally known as *resource allocation*. We consider the problem only in the message passing model.

Examples of resource allocation can be found in many applications including distributed database and file systems. In distributed database systems, a transaction can access multiple data records during its operation, and to maintain the consistency of the accessed data, each transaction usually locks the data records so that no other transactions can access them during its access. The locks on data records can

be considered as resources to which a transaction needs to have the exclusive access.

Resource allocation in these kinds of systems has been formulated as three kinds: *the dining philosophers problem*, *the drinking philosophers problem*, and *the dynamic resource allocation problem*. In all these problems, each process has a code segment singled out as *the critical region* where it uses the resources. A process enters its critical region after acquiring all its needed resources and then relinquishes them when exiting the region. If any two processes have overlapping resource requirements, we say that the two processes are in *conflict*. The correctness conditions of these problems dictate that no two conflicting processes can be in the critical region at the same time, and all processes requesting resources must eventually enter the critical region.

The dining philosophers problem [6] is a static version of the resource allocation problem, where the process set is fixed and each process requests a fixed set of resources periodically. A more dynamic version of the dining philosophers, called drinking philosophers [3], allows a process to request a different subset of its a priori given maximum resource requirement. Any two processes whose current resource requirements do not overlap may be in the critical region at the same time even though their maximum resource requirements overlap. In the dining and drinking philosophers problems, each process has a priori knowledge about its maximal set of conflicting processes. The dynamic resource allocation problem [1] is the most general of the three resource allocation problems. In the problem, the process set may dynamically change over time and each process may need, for its execution, any set of resources and may need different sets of resources at different times (i.e., a process has no prior knowledge about its maximum resource requirement and its conflicting processes).

The response time is used to measure the performance of resource allocation algorithms, which is defined to be the time difference between when a process requests resources and when it acquires them to enter the critical region. The message complexity is also used to measure the maximum number of messages required for a process to enter the critical region. These metrics are often expressed in terms of the following parameters.  $\delta$  is the maximum number of conflict-

ing processes at any time during the execution of the algorithm,  $r$  is the maximum number of resources that a process requests at a time,  $c$  is the maximum time that a process is in the critical region and  $d$  is the maximum message delay between any two processes.

### 1.1 Our results

We present a modular algorithm  $\mathcal{M}$  that uses any arbitrary resource allocation algorithm as a subroutine  $S$ . It can improve on the overall response time by letting each process wait, if it has to, only for its currently conflicting processes.

To be specific, consider a resource allocation algorithm of any kind with response time  $O(x \cdot (c + d))$  and message complexity  $O(m)$  for some  $x$  and  $m$ . If  $\mathcal{M}$  uses this algorithm as a subroutine  $S$ , the overall response time of the resulting algorithm is  $O(\delta c + \max\{\delta^2, x\}d)$  and its message complexity is  $\max\{m, O(r\delta)\}$ . Since  $c$  is multiplied by  $\delta$ , our modular algorithm reduces the impact of  $c$  on the worst case response time when  $x$  is bigger than  $O(\delta)$  (more precisely, bigger than  $3\delta$ ).

The modular algorithm  $\mathcal{M}$  uses the critical region of  $S$  (i.e., the exclusion property of the region) to schedule resource accesses in such a way that processes wait only for their currently conflicting processes. The critical region of  $S$  is not used to allocate actual resource, but to lockout competing processes while they schedule themselves for resource accesses. This modular construction effectively bounds the response time to be a function of  $\delta c$ , which is the lower bound. The time that a process spends in the critical region of  $S$  is also bounded to be  $O(d)$  in  $\mathcal{M}$ . Minimizing the term applied to  $c$  contributes to minimizing the overall actual response time of  $\mathcal{M}$  because the term is directly related to the level of concurrency allowed by the protocol (i.e., the number of processes in the critical region at the same time). In addition,  $c$  can often be large in most systems. For example, in distributed database systems, an access to each resource in the critical region requires disk accesses, and I/O latency is far (more than 20 times) greater than message delay in a typical local area network, more evidently in parallel processor systems.

Another advantage of our modular algorithm is that its concurrency is limited only by the current resource requirements of contending processes, but not by their maximum resource requirements. This allows our algorithm to be applicable to various resource allocation problems. For example, given any arbitrary dining philosophers algorithm as a subroutine, it behaves as a drinking philosophers algorithm, allowing more concurrency than the dining philosophers, and if the subroutine is a drinking philosophers algorithm or a dynamic resource allocation algorithm, so is the resulting algorithm. So, when we use Choy and Singh's *dining* philosophers algorithm [4] as a subroutine, which has response time  $O(\delta^2(c + d))$ , we obtain a *drinking* philosophers algorithm with improved response time  $O(\delta c + \delta^2 d)$  and message complexity  $O(r\delta)$ . This is the fastest known drinking philosophers algorithm with the stated message complexity (see Table 1). When we use Choy and Singh's dynamic resource allocation algorithm [5] as a subroutine, which has response  $O(\delta^2 c + (\delta^3 + \delta \log^* |\mathcal{U}|)d)$  with mes-

sage complexity  $O(\delta^2 + \delta \log^* |\mathcal{U}|)$  where  $\mathcal{U}$  is the universal set of from which process IDs are drawn<sup>1</sup>, we obtain a dynamic resource allocation algorithm with response time  $O(\delta c + (\delta^3 + \delta \log^* |\mathcal{U}|)d)$  and message complexity  $\max\{O(\delta^2 + \delta \log^* |\mathcal{U}|), O(r\delta)\}$ . When  $\log^* |\mathcal{U}|$  is less than  $\delta$ , our dynamic algorithm is the fastest known dynamic resource allocation algorithm with the stated message complexity (see also Table 1). A similar type of modularity was presented by Welch and Lynch [12], but unlike theirs, one modular algorithm can improve the performance of the subroutine being used.

The system structure of our solution is similar to those of Lynch [7] and Weidman et al. [13], in that there is one designated process for *each* resource, called a *resource manager*, that allocates its resource to the requesting processes or maintains information about conflicting processes. The fact that the resource managers also participate in scheduling accesses to resources does not make our algorithm more centralized than other algorithms in terms of fault-tolerance and performance bottlenecks. Since each resource manager deals with resource requests only pertaining to its own resource, a crash of a resource manager doesn't necessarily affect the progress of the other processes that do not require the resource of the crashed resource manager. In fact, the effect of the failure of one resource manager can be less severe than that of the failure of one process (that is not a resource manager). To see this, imagine that a process crashes in the critical region while holding a set of resources. In this situation, all the processes that require these resources cannot progress. This has the same effect as the failure of all the resource managers that manage these resources. Therefore, in terms of fault-tolerance, our solution is no less distributed than *fully distributed* solutions, such as those in [1, 2]. In terms of performance bottleneck, our algorithm gives an improved worst case response time than other priorly known algorithms.

We simulated our combined algorithms and other known dining philosophers algorithms using a discrete event simulation technique. The simulation results indicate that the average performance (i.e., response time and message complexity) of our algorithm can be remarkably better than the other known algorithms, especially when resource contention among processes is high and the average time that a process remains in the critical region is larger than the average message delay.

### 1.2 Previous results

Dijkstra [6] first modeled the resource allocation problem as a ring of 5 processes, called the dining philosophers, where each process shares a resource with each neighbor. Later, Lynch [7] generalized the problem to an arbitrary conflict graph where a node represents a process and an edge represents a sharing of resources between two processes. Lynch's solution uses an edge coloring algorithm to set a partial ordering on the shared resources, so that each process requests its needed resources in that order. The response time is  $O(c^\delta(c + d))$  where  $c$  is the number of colors used in the coloring. The message complexity is  $O(\delta)$ .

<sup>1</sup>  $\log^* n = \min\{i : \log^i n \leq 1\}$ .

**Table 1.** Time and message complexities of resource allocation algorithms

Authors	Problem	Time	Message
Lynch [7]	dining	$O(c^\delta(c+d))$	$O(\delta)$
Styer and Peterson [11]	dining	$O(\delta^{\log \delta+1}(c+d))$	$O(\delta^{\log \delta+1})$
Choy and Singh [4]	dining	$O(\delta^2(c+d))$	$O(\delta)$
Page et al. [9]	dining	$O(\delta c + nd)$	$O(\delta^2)$
Chandy and Misra [3]	drinking	$O(n(c+d))$	$O(n)$
Rhee	drinking	$O(\delta c + \delta^2 d)$	$O(\max\{\delta^2, r\delta\})$
Awerbuch and Saks [1]	dynamic	$O(\delta c + \delta^2 \log  \mathcal{L} d)$	$O(\delta^2 \log  \mathcal{L} )$
Weidman et al. [13]	dynamic	$O(n(c+d))$	$O(\delta)$
Choy and Singh [5]	dynamic	$O(\delta^2 c + (\delta^3 + \delta \log^*  \mathcal{L} )d)$	$O(\delta^2 + \delta \log^*  \mathcal{L} )$
Rhee	dynamic	$O(\delta c + (\delta^3 + \delta \log^*  \mathcal{L} )d)$	$O(\max\{\delta^2 + \delta \log^*  \mathcal{L} , r\delta\})$

Styer and Peterson [11] extended Lynch’s algorithm to develop a dining philosophers algorithm with response time  $O(\delta^{\log \delta+1}(c+d))$  and message complexity  $O(\delta^{\log \delta+1})$ . Page, Jacob and Chern [9] presented a dining philosophers algorithm with response time  $O(\delta c + nd)$  and message complexity  $O(\delta^2)$ . Choy and Singh [4] developed a dining philosophers algorithm with response time  $O(\delta^2(c+d))$  and message complexity  $O(\delta)$ . They also include some discussion of fault-tolerance.

Chandy and Misra [3] presented a dining philosophers algorithm using an acyclic directed version of the conflict graph. They also first proposed the drinking philosophers problem and its solution which uses their dining philosophers solution as a subroutine. The response time for both of the solutions are  $O(n(c+d))$ , and the message complexity is  $O(\delta)$ . Welch and Lynch [12] generalized the modular construction of Chandy and Misra’s drinking philosophers algorithm to come up with a drinking philosophers algorithm which uses, as a subroutine, any dining philosophers algorithm. Its response time and message complexity are equal to those of the subroutine.

Awerbuch and Saks [1] first defined and solved the dynamic resource allocation problem. In their model, they assume that processes have to know a priori the IDs of their conflicting processes.<sup>2</sup> The algorithm’s worst case response time is  $O(\delta c + \delta^2(\log |\mathcal{L}|)d)$ , and the message complexity is  $O(\delta^2 \log |\mathcal{L}|)$ . Weidman et al. [13] developed a dynamic resource allocation algorithm using Chandy and Misra’s drinking philosophers algorithm as a subroutine. Its response time and message complexity are the same as those of Chandy and Misra’s. Bar-Ilan and Peleg [2] developed a *synchronous* algorithm that improves on Awerbuch and Saks’ algorithm to have response time  $O(\delta c + \delta(\log |\mathcal{L}|)d)$  in a *synchronous* network. Choy and Singh [5] also developed a dynamic resource allocation algorithm with worst case response time  $O(\delta^2 c + (\delta^3 + \log^* |\mathcal{L}|)d)$  and message complexity  $O(\delta^2 + \delta \log^* |\mathcal{L}|)$ .

## 2 The system model

There exists a (finite or infinite) set of processes  $P = \{p_1, p_2, p_3, \dots\}$ . Processes communicate by passing messages. There are three types of process steps: *send*, *receive*

<sup>2</sup> Note that our definition of the dynamic resource allocation problem is more general in that it doesn’t assume a priori knowledge about the conflict processes.

and *local step*. Send and receive are communication primitives and a local step changes local variables of processes. Each process  $p_i$  is modeled by a finite state automaton with state set  $Q_i$ . The state set  $Q_i$  includes an initial state  $q_{0,i}$ .

The automaton for each process is specified by a single guarded command set  $[B_1 \rightarrow A_1 \square B_2 \rightarrow A_2 \square \dots \square B_m \rightarrow A_m]$ . Each  $B_i \rightarrow A_i$  is a guarded command, where a guard  $B_i$  is either a boolean expression or a message reception (receive step), or a conjunction of both, and a finite list of action statements  $A_i$  that consists of either multiple local steps or one send step, or both. While the execution of  $A_i$  appears atomic to all the other processes, the statements within  $A_i$  will be executed in sequence.

Processes communicate by sending messages to each other. A send step represents the sending of message to a process, and a receive step of a process involves a reception of one message destined to the process. We assume that messages sent are eventually received by their destination processes within a finite time.

A *configuration* is a vector  $C = \{q_1, q_2, \dots\}$  where  $q_i$  is the local state of  $p_i$  for each  $p_i \in P$  (including the network). A guarded command is *enabled* in a configuration if its associated boolean expression is true and associated receive, if any, can return nonempty messages, i.e., the messages specified in the receive are in the buffer of its process. An *execution of a guarded command* involves an atomic execution of all the steps in the action statements of the guarded command. It results in simultaneous changes to the state of the process of the guarded command based on the previous state of the process, and possibly to the state of the network if the guarded command involves a send step. A guarded command enabled in a configuration  $C$  can be *applied* to  $C$  to yield a new configuration  $C'$  as a result of the execution of the guarded command.

A *system* is specified by describing  $P$ , an initial configuration  $C_0 = (q_{0,1}, q_{0,2}, \dots)$ , and the automaton of all processes in  $P$ . An *execution sequence* of a system is an infinite alternating sequence of configurations and enabled guarded commands  $C_0, \pi_1, C_1, \pi_2, \dots, C_i, \pi_i, \dots$ , where  $\pi_i$  is an enabled guarded command in  $C_{i-1}$ ; and  $C_i$  is obtained by applying  $\pi_{i-1}$  to  $C_{i-1}$ . We say that in an execution sequence, a guarded command is *continually enabled* from  $C_i$  to  $C_j$ ,  $i < j$ , if the guarded command is enabled in every configuration from  $C_i$  to  $C_j$  and is not applied to any configuration in between  $C_i$  and  $C_{j-1}$ . We also say that in an execution sequence  $\sigma$ , a guarded command  $g$  is *enabled before* a guarded command  $g'$  in  $C_k$  if there exists a sequence of configurations  $C_i, \dots, C_j, \dots, C_k$  in  $\sigma$  such

that  $g$  is continually enabled from  $C_i$  to  $C_k$ , but  $g'$  is not continually enabled from  $C_i$  to  $C_j$ .

An *execution* is an execution sequence satisfying the following fairness conditions: (1) if  $\pi_i$  is a guarded command of process  $p$  (that is not the network) and applied to  $C_i$ , then there is no guarded command of  $p$  that is enabled before  $\pi_i$  in  $C_i$  (i.e. the FIFO execution); (2) all continually enabled guarded command will be executed eventually.

A *timed execution*  $(\alpha, T) = C_0, (\pi_1, t_1), \dots, (\pi_j, t_j), \dots$  satisfies the following conditions: (1)  $\alpha = C_0, \pi_1, C_1, \pi_2, \dots, [[C_i, \pi_{i+1}]] \dots$  is an execution; (2)  $T$  is a mapping from guarded commands to real numbers that associates a real time with each guarded command in the execution. The sequence  $t_0, t_1, \dots, t_i \dots$  is nondecreasing and unbounded. (3) All messages sent are received in finite time.

If  $V$  is a state variable of a process and  $t$  is a real number,  $V(t)$  denotes the value of  $V$  in the configuration  $C_j$  where  $T(\pi_j) \leq t < T(\pi_{j+1})$ , i.e., a configuration  $C_j$  represents the states of the system during time interval  $[T(\pi_j), T(\pi_{j+1}))$ .

Note that the system model here is completely asynchronous because there are no constraints on relative timing of process steps and message delays.

### 3 Resource allocation problems

We now specialize the general system model in Sect. 2 for resource allocation problem. Let  $R$  be the set of resource in the system. There exists a set of processes  $U (\subseteq P)$  called the *users* that need subsets of  $R$  for their execution at various times.

Let  $R_i(t)$  be the *resource requirement* of user  $i$  at time  $t$ . Let  $R_i^{max}$  be the *maximum resource requirement* of a user  $i$  such that for every execution  $\alpha$ ,  $R_i^{max}$  is the union of the resource requirements of user  $i$  in  $\alpha$ .

Each user's local states are partitioned into four regions. In the *trying region*, the user requests its required resources. Having acquiring the resources, the user enters the *critical region*. It remains in the region for a finite time using the resources. When the user is finished with the resources, it enters the *exit region*, where it relinquishes the resources. Otherwise, the user is in the *remainder region*. To specify this, we assume that each user has a local variable, called *region*, whose value is set to *Trying*, *Critical*, *Exit*, or *Remainder* if and only if the user is in the trying region, the critical region, the exit region or the remainder region respectively. Initially, every user is in the remainder region.

At some time  $t$ , if users  $i$  and  $j$  are in the trying or critical region, and  $R_i(t) \cap R_j(t) \neq \emptyset$ , then we say that user  $i$  *conflicts* with user  $j$  at time  $t$ .

A *dining philosophers* algorithm is a system with a finite and fixed set  $U$  where  $\forall i \in U$ ,  $R_i(t) = R_i^{max}$  and  $R_i^{max}$  is a priori known to all processes. Each user's code is well-formed, and the algorithm must satisfy the following two conditions: (1) (*exclusion*) in any execution of the algorithm, if users  $i$  and  $j$  are both in the critical region at time  $t$ , then  $i$  and  $j$  do not conflict with each other; (2) (*no-lockout*) in any execution of the algorithm, if a user is in the trying region or in the exit region, then it leaves its current region in finite time assuming no user remains in the critical region forever.

In a drinking philosophers algorithm and a dynamic resource allocation algorithm, users can have more concurrency than in a dining philosophers algorithm because users only need their current resource requirement to be satisfied, but not their maximums. To formalize this concept, we define the following condition: (3) (*concurrency*) in any execution of the algorithm, if there is no conflicting user of user  $i$  in its trying region after and when  $i$  enters its trying region, then user  $i$  eventually enters the critical region. This condition is stronger than the no-lockout condition, that is, as long as there are no conflicting users while user  $i$  is in the trying region, user  $i$  is never stuck in the trying region *even if other users are using other resources forever*. The same concurrency condition is also given in [12].

A *drinking philosophers* algorithm is a system with a finite and fixed user set  $U$  where for all users  $i$ ,  $R_i^{max}$  is a priori known to all processes, but  $R_i(t)$  is a priori unknown. Each user's code is well-formed, and the algorithm has to satisfy the exclusion, no-lockout and concurrency conditions.

A *dynamic resource allocation* algorithm is a system with an infinite set  $U$  where for all users  $i$ ,  $R_i^{max}$  and  $R_i(t)$  are a priori unknown to all processes. Each user's code is well-formed, and the algorithm also has to satisfy the exclusion, no-lockout and concurrency conditions.

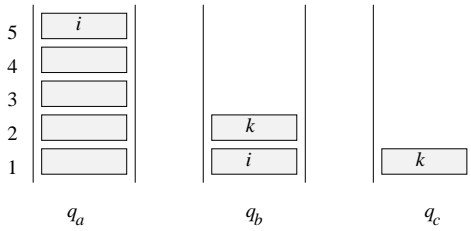
## 4 Algorithm

### 4.1 Informal description

The basic idea of our algorithm is an implementation of a distributed queue, where each user in the trying region has a position in the queue and enters the critical region in the order of its position. The distributed queue runs under the following operational rules: (1) while a user is in the trying region, it occupies a position in the queue in such a way that no two conflicting users occupy the same position; (2) users at the front of the queue enter the critical region and leave the queue when they enter the exit region; (3) when all the users at the front of the queue leave the queue, all the other users in the queue advance one position, preserving their relative order and the first rule; and (4) a newly joining user does not prevent any users in the queue from advancing to their next positions.

It can be proved that the distributed queue with these rules guarantees no-lockout and exclusion. Since no two conflicting users occupy the same position in the queue and only the one at the front of the queue enters the critical region, rules 1 and 2 ensure that no two conflicting users are in the critical region at the same time (exclusion). By rule 2, when a user finishes in the critical region, it leaves the queue, and by rule 3, all users in the queue will advance one position. Furthermore rule 4 ensures that even if some user enters the queue just before this advancement, all the users in the queue advance one position. It prevents the user that has just left the queue and enters the queue again from obstructing the advancing users. Thus, rules 3 and 4 ensure the no-lockout condition. (Formal proof is given in Sect. 5).

In our implementation, there is a group of special processes, called *resource managers*, each assigned to one resource. The distributed queue mentioned above is implemented by having one queue per resource manager. Each



**Fig. 1.** An illustration of queues when a user selects the end of position of queues: user  $k$  has to wait for all the users that  $i$  is waiting for (occupied positions are shaded)

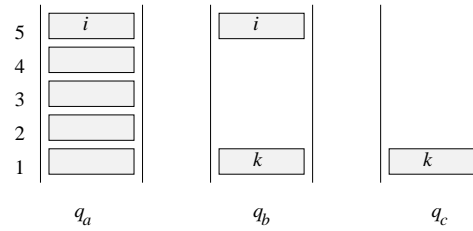
resource manager  $rm_k$  maintains a queue  $q_k$ . The goal is to make these queues as a whole behave as the distributed queue by enforcing the above-mentioned four rules. The following discusses our implementation.

We apply a modular approach to implement the first rule: we use another resource allocation algorithm (of any kind) as a subroutine. Each user  $i$  in the trying region at time  $t$  first runs the subroutine. After entering the critical region of the subroutine, it selects a position to occupy in each  $q_k, k \in R_i(t)$ . The idea is that while a user  $i$  is in the critical region of the subroutine where  $i$  is selecting a position in the queue, no conflicting users are in that region. Therefore, no two conflicting users can select the same position in the same queue.

In order to select a unique position in a queue, user  $i$  sends a *report* message to the resource manager of the queue, and the resource manager acknowledges with a *marked* message that contains information about positions occupied by other users in the queue, based on which  $i$  selects a position in each  $q_k$ . After finishing the selection, the user informs all the relevant resource managers of the selected positions with a *select* message, and then it leaves the critical region of the subroutine. It is easy to see that it takes at most  $O(d)$  time for a user to select the positions after entering the critical region of the subroutine. If the response time of the subroutine is  $X(C + d)$  for some  $X$ , a user can select its position in time  $O(Xd)$ .

The position that a user selects in a queue affects the overall response time of the user. For example, a user may select the tail position of each queue. But this can cause unnecessarily long response time. To see this, suppose that a user  $i$  needs resources  $a$  and  $b$ ,  $q_a$  is occupied by four users, and  $q_b$  is empty. If user  $i$  selects the end position of each queue, it will occupy position 5 in  $q_a$  and position 1 in  $q_b$ . Suppose that a user  $k$  requiring resources  $b$  and  $c$  subsequently selects a position 2 in  $q_b$  and position 1 in  $q_c$ . Now,  $k$  has to wait for  $i$  to leave the queue which is in turn waiting for four other users to finish (see Fig. 1). This waiting chain can grow up to  $\Omega(n)$ -users in length, which results in  $\Omega(nc)$  worst case response time.

One reasonable approach is to have each user  $i$  to select the smallest unoccupied position in every  $q_k, k \in R_i(t)$ . That is, a user looks for a “hole” in every queue. In the above example,  $i$  will select position 5 because position 5 is the smallest position unoccupied in both  $q_a$  and  $q_b$ , and then  $k$  will select position 1 because position 1 is unoccupied in  $q_b$  and  $q_c$ . Therefore,  $k$  does not have to wait for  $i$  to finish (see Fig. 2). This way, each user, if it has to, waits only



**Fig. 2.** An illustration of queues when a user selects a “hole”: user  $k$  does not wait for the users that user  $i$  is waiting for

for its conflicting users. Later, this approach will be slightly modified to accommodate the fourth rule.

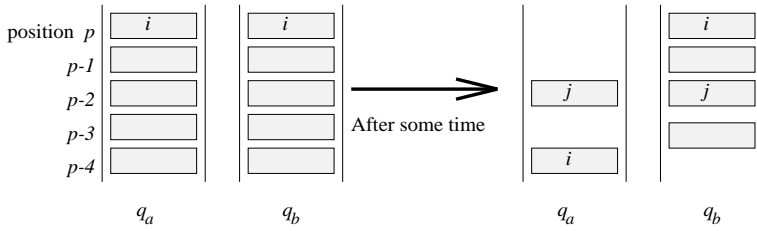
To implement the second rule, each resource manager sends a *grant* message to the user at the front of its queue. A user  $i$ , when receiving a *grant* message from every  $rm_k, k \in R_i(t)$ , enters the “real” critical region (not of subroutine). After it is finished in its critical region, it sends a *release* message to every  $rm_k$ . This is the exit region of user  $i$ . Upon receiving a *release* message, a resource manager marks the front of the queue unoccupied. If a position in a queue becomes unoccupied, any user occupying the immediately succeeding position needs to advance.

Now, we need to implement the third rule in which, when all the users at the front of the queue leave the queue, all the other users in the queue advance one position, preserving their relative order and the first rule. Because of the asynchrony in the system, arbitrary advancement of users in a queue may break the relative order among conflicting users. To illustrate this, suppose that user  $i$  selects a position  $p$  in resources  $a$  and  $b$ . Assume that all the users at the positions less than  $p$  in  $q_a$  finish the critical region and leave the queue before those in  $q_b$ . Suppose that after some time, the positions of user  $i$  become equal to, say,  $p - 4$  in  $q_a$  and  $p$  in  $q_b$ . It is possible that not all positions below position  $p$  in  $q_b$  are occupied. So, in the meanwhile, some user  $j$  which also requires resources  $a$  and  $b$  selects, say, position  $p - 2$  in both  $q_a$  and  $q_b$ . Now, because  $j$  waits for  $i$  in  $q_a$  while  $i$  waits for  $j$  in  $q_b$ , there is no relative order between  $i$  and  $j$ , and this causes deadlock (see Fig. 3).

To implement the third rule without breaking the relative order among conflicting users, we let each resource manager  $rm_k$  send a *dec* message to user  $i$  when the position immediately preceding user  $i$ 's position in  $q_k$  is unoccupied. When user  $i$  receives one *dec* message from each  $rm_k, k \in R_i(t)$ , it sends an *advance* message to every  $rm_k$ . Upon receiving the *advance* message,  $rm_k$  advances user  $i$ 's position in  $q_k$  to the next lower position.

It can be proved that when the above scheme is used, the difference between the positions of user  $i$  in any two queues is less than two, i.e., the positions of user  $i$  in the different queues, advance almost “in synchrony”. The proof is given in Sect. 5. This preserves the relative order among the conflicting users. For example, in the above example, user  $i$  will remain in position  $p$  or  $p - 1$  in both  $q_a$  and  $q_b$ . Therefore, user  $j$  cannot occupy a position in between the two positions of user  $i$ .

The fourth rule prevents new users from interfering with the advancement of the users in the queues. It can be easily implemented by dictating those new users not consider



**Fig. 3.** An illustration of queues when a user  $i$  selects position  $p$  initially and then after some time, user  $j$  selects  $p-2$ . This shows that a careless advancement of users (here user  $i$  in  $q_a$ ) causes a deadlock

```

01 □  $region = Trying$  and  $sub-region = Remainder \rightarrow$ 
02    $sub-region := Trying;$ 
03    $subR_i := R_i;$ 
04 □  $sub-region = Critical$  and  $req-report = false \rightarrow$ 
05    $req-report := true;$ 
06   for all  $j \in R_i$ : send  $report_{i,j}$ ;
07 □ Receive  $marked_{j,i}, \forall j \in R_i \rightarrow$ 
08    $p := \min\{\mathcal{N} - \bigcup_{j \in R_i} marked_{j,i}\};$ 
09   for all  $j \in R_i$ : send  $select(p)_{i,j}$ ;
10    $req-report := false;$ 
11    $sub-region := Exit;$ 
12 □ Receive  $grant_{j,i}, \forall j \in R_i \rightarrow$ 
13    $region := Critical;$ 
14 □ Receive  $dec(p)_{j,i}, \forall j \in R_i$  for some  $p \rightarrow$ 
15   for all  $j \in R_i$ : send  $advance(p)_{i,j}$ ;
16 □  $region = Exit \rightarrow$ 
17    $region := Remainder;$ 
18   for all  $j \in R_i$ : send  $release_{i,j}$ ;

```

**Fig. 4.** Code for user  $i$

the positions immediately preceding any occupied ones in the queues of their required resources when they select the initial positions. This implementation of the fourth rule is somewhat related to the first rule. Imagine a situation where a user  $i$  sends an *advance* message to a resource manager  $rm_k$  so that  $i$  can advance to  $p-1$  from  $p$ . Without such implementation of the fourth rule, it is possible for a new user  $j$  to occupy position  $p-1$  in  $q_k$  just before  $rm_k$  receives the *advance* message. Now, we have a situation that either both  $i$  and  $j$  may occupy the same position in  $q_k$ , or  $i$  may not be able to advance to position  $p-1$  at all. Thus, by preventing a new user from occupying any positions immediately preceding already occupied positions, we can keep new users from obstructing the advancement without violating the first rule.

We obtain the time complexity of our protocol using the following argument. The initial position of a user is always less than  $O(\delta)$ , because (1) each user selects the minimum position that is not occupied by all its conflicting users; (2) the difference between any two positions of user  $i$  in queues is less than two (which means user  $i$  occupies no more than two position in queues at any time); and (3) there are at most  $\delta$  conflicting users at any time. By induction on the position that a user occupies, it can be shown that a user advances one position in time  $O(\delta d)$  after all position 1's in all queues are unoccupied (which happens at every  $O(c+d)$  time in the worst case). The proof is given in Sect. 5. Therefore, since there are at most  $O(\delta)$  positions in front of any user, a user will reach the front of the queue in time  $O(\delta c + \delta^2 d)$ . Since it takes  $O(Xd)$  time for a user to select an initial position, the overall response time of the algorithm is  $O(\delta c + \delta^2 d + Xd)$ .

For the message complexity, before a user  $i$  advances to a new position in  $q_k$ , user  $i$  receives at most  $r$  *dec* messages and sends at most  $r$  *advance* messages while user  $i$  is at position  $p$ . Therefore, because the initial position is always

in  $O(\delta)$ , the message complexity is  $O(M + r\delta)$  if  $M$  is the message complexity of the subroutine.

## 4.2 Formal description

As described in Sect. 3, in each user's code, *region* is set to *Trying*, *Critical*, *Exit* or *Remainder* if and only if a user is the trying region, the critical region, the exit region or the remainder region. The subroutine also has its own "region" variable. To distinguish these two variables, we call *region* in the subroutine *sub-region*.

Given a subroutine, we take only the resource allocation part of the code and concatenate it to the user code described in Fig. 4. All the occurrences of *region* in the resource allocation part of the subroutine are replaced by *sub-region*, and the variable  $R_i$  in the subroutine is also replaced by  $subR_i$ . It is assumed that the subroutine is well-formed.

The following messages and state variables (in addition to ones described in above) are used in the algorithm described in Figs. 4 and 5.

- $select(p)_{i,k}$ : message from user  $i$  to  $rm_k$ ; indicates that  $i$  selected position  $p$  in  $q_k$ .
- $report_{i,k}$ : message from user  $i$  to  $rm_k$ ; requests information about the occupied positions in  $q_k$ .
- $release_{i,k}$ : message from user  $i$  to  $rm_k$ ; indicates that it has finished using the resource.
- $marked_{k,i}$ : message from  $rm_k$  to user  $i$ ; contains all the occupied positions and their preceding positions in  $q_k$ .
- $grant_{k,i}$ : message from  $rm_k$  to user  $i$ ; indicates that resource  $k$  is granted to the user.
- $dec(p)_{k,i}$ : message from  $rm_k$  to user  $i$ ; indicates that position  $p-1$  of  $q_k$  is unoccupied.
- $advance(p)_{i,k}$ : message from user  $i$  to  $rm_k$ ; indicates that  $rm_k$  can advance user  $i$  into position  $p-1$ .
- *req-report*: a boolean variable; true when a user sent *report* messages, but hasn't received *marked* messages.
- *has\_received\_advanced(p)*: a boolean variable; becomes true when  $rm_k$  receives  $advance(p)_{i,k}$  and becomes false when user  $i$  advances to position  $p-1$  (i.e., *advance\_one\_position(p)* is called).
- *occupant<sub>k</sub>(p)*: a variable; contains the ID of the user that occupies position  $p$  of  $q_k$ . 0 if the position is unoccupied. (We assume there is no user with ID zero.)
- *rm\_critical<sub>k</sub>*: a boolean variable; true when  $rm_k$  received a *report* message from some  $i$ , but hasn't received a *select* message from  $i$ . It indicates that  $i$  is in the critical region of the subroutine.
- *has\_dec\_sent(p)*: a boolean variable; true when a *dec(p)* is sent, and false when a user occupies position  $p$ .

```

01  $\square$   $\neg rm\_critical_k$ , receive  $report_{i,k} \rightarrow$  // received report msg.
02    $marked_{k,i} := \{j - 1, j : \forall j, occupant_k(j) \neq 0\}$ ;
    // marked contains information about all
    // occupied positions and their immediately preceding positions.
03   send  $marked_{k,i}$ ;
04    $rm\_critical_k := true$ ; // enter critical section of the subroutine.
05  $\square$   $rm\_critical_k$ , receive  $select(p)_{i,k} \rightarrow$  // finished the selection
06    $rm\_critical_k := false$ ; // leave critical section of the subroutine.
07    $occupant_k(p) := i$ ;
08   if  $p = 1$  then send  $grant_{k,i}$ ; // send a grant msg if  $i$  is at the front.
09   adjust_queue( $p$ );
10  $\square$   $\neg rm\_critical_k$ , receive  $advance(p)_{i,k} \rightarrow$  // recvd an advance msg
11    $has\_received\_advance(p) := true$ ;
12   adjust_queue( $p$ );
13  $\square$   $\neg rm\_critical_k$ , receive  $release_{i,k} \rightarrow$  // release the res. to the next user.
14    $occupant_k(1) := 0$ ;
15   adjust_queue(2);

16 Procedure adjust_queue( $p$ )
17   while ( $occupant_k(p) \neq 0$ ) and ( $occupant_k(p - 1) = 0$ )
18     if ( $has\_dec\_sent(p) = false$ ) then
19        $has\_dec\_sent(p) := true$ ;
20       send  $dec(p)_{k,occupant_k(p)}$ ;
21     endif
22     if ( $has\_received\_advance(p) = true$ ): advance_one_position( $p$ );
23      $p := p + 1$ ;
24   end while

25 Procedure advance_one_position( $p$ )
26    $occupant_k(p - 1) := occupant_k(p)$ ; // advance one position
27    $occupant_k(p) := 0$ ; // reinitialize position variables
28    $has\_received\_advance(p) := false$ ;
29    $has\_dec\_sent(p) := false$ ;
30   if ( $p - 1 = 1$ ) then send  $grant_{k,occupant_k(p-1)}$ ;
    // if the next position is empty, send a dec message.
31   if ( $has\_dec\_sent(p - 1) = false$ ) and ( $occupant_k(p - 2) = 0$ ) then
32      $has\_dec\_sent(p) := true$ ;
33     send  $dec(p - 1)_{k,occupant_k(p-1)}$ ;

```

Fig. 5. Code for resource manager  $rm_k$

## 5 Correctness proof and performance analysis

For convenience of presentation, we define some terms. We denote by  $i_k$  the fact that user  $i$  occupies a position in  $q_k$ . If a user  $i$  occupies or advances to a position  $p$  in  $q_k$ , then we say that  $i_k$  occupies or advances to  $p$ . If  $rm_k$  sends a  $dec(p)$  for any  $p$  while  $i$  occupies position  $p$  in  $q_k$ , then instead of saying that the resource manager sends the  $dec$  messages, we say more conveniently that  $i_k$  sends  $dec(p)$ . We also assume that each user enters the trying region no more than once. This assumption does not affect the correctness of our protocol as the protocol relies only on the current resource requirement of users.

As it is straightforward to show that the user code is well-formed, we only prove the exclusion, no-lockout and concurrency conditions.

**Lemma 5.1.** *In any timed execution of the algorithm, for any conflicting users  $i$  and  $j$ ,  $i_k$  and  $j_k$  do not occupy the same position at the same time.*

*Proof.* Without loss of generality, we assume that  $i$  selects a position in  $q_k$  after  $j$  does (i.e.,  $i$  enters the critical region of the subroutine after  $j$ ). Note that they cannot select positions at the same time because of the exclusion condition of the subroutine and because  $i$  conflicts with  $j$ .

When  $i$  selects a position in  $q_k$ , it receives a  $marked$  message from  $rm_k$ , which contains information about all the occupied positions in  $q_k$ .  $rm\_critical_k$  in the resource

manager's code is set to true when  $rm_k$  receives a  $report$  message from user  $i$  (see line 4 in Fig. 5), which is sent only after the user enters the critical region of the subroutine. Since other conflicting users cannot be in the region at the same time,  $rm_k$  doesn't receive a  $select$  message until  $i$  sends it. Thus, while  $i$  is selecting a position in  $q_k$ ,  $rm\_critical_k$  remains true until  $select_{i,k}$  is received. Since  $rm_k$  doesn't receive any other message while  $rm\_critical_k$  is true, the positions of other users in  $q_k$  do not change during the selection. Also by the code,  $i_k$  doesn't select any position occupied by  $j_k$  or its preceding position (see line 8 in Fig. 4). This guarantees that  $i$  never selects the position that  $j_k$  selected since user  $i$  selects a position that are unoccupied in  $q_k$ .

Since  $i_k$  advances to a new position only after its immediately preceding position is unoccupied (see lines 17 and 22 in Fig. 5), it never happens that  $i_k$  advances to the position that  $j_k$  occupies or vice versa. ■

**Theorem 5.2.** (Exclusion) *In any timed execution of the algorithm, if users  $i$  and  $j$  are both in the critical region, then  $i$  and  $j$  do not conflict with each other.*

*Proof.* By Lemma 5.1,  $i_k$  and  $j_k$  never occupy the same position at the same time. By the code, only the users at position 1 will be in the critical region (see lines 8 and 30 in Fig. 5). Thus, if users occupy different positions in a queue, there is no such case that they are in the critical region at the same time. ■

We now prove the no-lockout condition of the algorithm. The proof is structured as follows. We first prove in Lemma 5.5 (using Claims 5.4 and 5.3) that a user  $i_k$  advances one position from a position  $p$  if and only if position  $p - 1$  in every  $q_{k'}$ ,  $k' \in R_i$ , is unoccupied. The lemma implies that all  $i_k$ 's are advancing "in synchrony" whenever position  $p - 1$  in every  $q_{k'}$ ,  $k' \in R_i$ , is unoccupied. As every  $i_k$  initially occupies the same position in every  $q_k$ , the distance between any  $i_k$  and  $i_{k'}$  ( $k \neq k'$ ) is at most 1.

Then we prove in Lemma 5.7 that if a user  $j_k$  occupies a position lower than  $i_k$ , it is always the case that  $j_{k'}$  occupies a position lower than  $i_{k'}$ , e.g., there is no such situation that while a user  $i_k$  is at position 2 and a user  $j_k$  is at position 1,  $i_{k'}$  is at 1 and  $j_{k'}$  is at 2. Lemmas 5.5 and 5.7 together guarantee that all users that occupy position 1 at all queues will eventually receive a *grant* message from their resource managers, enter the critical region, and leave the queues. Also, using Lemma 5.5, Lemma 5.9 proves that after all users that occupy position 1 at any queue leave the queues, all other users in the queues advance one position eventually. Inductively applying this argument, we can prove that all users in the queue eventually advance to position 1. This proves the no-lockout condition.

*Claim 5.3.* If  $\text{has\_dec\_sent}(p)_k$  is true and  $i_k$  is at  $p$ , then  $rm_k$  sends at least one *dec*( $p$ ) to  $i$ .

*Proof.* Since only at line 19 in Fig. 5,  $\text{has\_dec\_sent}(p)_k$  becomes true, after which a *dec*( $p$ ) message is sent to the occupant of  $p$ , we only need to prove that when  $i_k$  first occupies  $p$ ,  $\text{has\_dec\_sent}(p)_k$  is false.

Assume by way of contradiction that when  $i_k$  first occupies  $p$ ,  $\text{has\_dec\_sent}(p)_k$  is already true. Since  $\text{has\_dec\_sent}(p)_k$  is initially set to false, there must have been some user  $j_k$  that occupied  $p$  and set  $\text{has\_dec\_sent}(p)_k$  to true (at line 19 in Fig. 5) previously, and some other user  $s_k$  that subsequently occupied  $p$  (at either line 7 or line 26 in Fig. 5) and found  $\text{has\_dec\_sent}(p)_k$  to be true. Note that  $\text{has\_dec\_sent}(p)_k$  becomes false only at line 28 in Fig. 5 which is executed when the occupant of  $p$  advances to  $p - 1$  (line 26 in Fig. 5). Therefore, the fact that  $\text{has\_dec\_sent}(p)_k$  was already true when  $s_k$  first occupied  $p$  implies that  $j_k$  has not had moved from  $p$  yet. This contradicts Lemma 5.1 since  $j_k$  and  $s_k$  were at the same position in the same queue. ■

*Claim 5.4.* An  $rm_k$ ,  $k \in R_i$  sends at least one *dec*( $p$ ) message to user  $i$  if and only if position  $p - 1$  in  $q_k$  is unoccupied and position  $p$  in  $q_k$  is occupied by  $i$ .

*Proof.* (**If part**) The condition that position  $p - 1$  in  $q_k$  is unoccupied and position  $p$  in  $q_k$  is occupied by  $i$  becomes true only either (1) when  $p - 1$  becomes unoccupied while  $i_k$  is at  $p$ , or (2) when  $i_k$  first occupies position  $p$ ,  $p - 1$  is already unoccupied.

Case 1 happens only either when  $rm_k$  receives an *advance*( $p - 1$ ) message or when  $p = 2$  and  $rm_k$  receives a *release* message. In either case,  $\text{adjust\_queue}(p)$  is called. Thus, line 18 in Fig. 5 will be executed

Case 2 happens only either when  $i$  selects position  $p$  by sending a *select*( $p$ ) or when  $i_k$  advances to  $p$  from  $p + 1$ . When  $rm_k$  receives a *select*( $p$ ) message, it calls

$\text{adjust\_queue}(p)$  (line 9 in Fig. 5). When  $i_k$  advances to  $p$  from  $p + 1$ , it is always checked whether  $p - 1$  is occupied or not. Line 31 in Fig. 5 will be executed.

In all cases,  $\text{has\_dec\_sent}(p)_k$  is checked and if it is not set to true, a *dec*( $p$ ) message is sent to  $i$ . If it is set to true, then by Claim 5.3, at least one *dec*( $p$ ) message must be sent to  $i$ .

(**Only if part**) Trivially true by the code (see lines 17 and 31 in Fig. 5). ■

We say that a user  $i$  is *aligned* at position  $p$  if for every  $k$ ,  $k \in R_i$ ,  $\text{occupant}_l(p) = i$ .

**Lemma 5.5.** An  $rm_k$ ,  $k \in R_i$  executes  $\text{advance\_one\_position}(p)$  if and only if  $i$  is aligned at position  $p$ , and for every  $l$ ,  $l \in R_i$ ,  $\text{occupant}_l(p - 1) = 0$ .

*Proof.* By Claim 5.4, user  $i$  receives a *dec*( $p$ ) message from all the  $rm_l$ 's if for all  $l$ ,  $l \in R_i$ ,  $\text{occupant}_l(p - 1) = 0$  and  $\text{occupant}_l(p) = i$ . By line 15 in Fig. 4, user  $i$  will send an *advance*( $p$ ) message to  $rm_k$ . Receiving the message,  $rm_k$  sets  $\text{has\_received\_advance}(p)$  to true and  $\text{adjust\_queue}(p)$  is called in which line 22 in Fig. 5 is executed.

$\text{has\_received\_advance}(p)$  is set to true only at line 11 which is when  $rm_k$  receives *advance*( $p$ ).  $\text{has\_received\_advance}(p)$  is set to false either initially or when a user at  $p$  advances to  $p - 1$  (at line 28). Thus if  $\text{has\_received\_advance}(p)$  is true, then it has received an *advance*( $p$ ) from the current occupant of position  $p$ . Note that by the code, an *advance*( $p$ ) message is sent by the occupant of  $p$  only when the occupant receives a *dec*( $p$ ) message from all the resource managers of the resources it requires. Thus, by Claim 5.4, the fact that  $\text{has\_received\_advance}(p)$  is true implies that for all  $l$ ,  $l \in R_i$ ,  $\text{occupant}_l(p - 1) = 0$  and  $\text{occupant}_l(p) = i$ . ■

Lemmas 5.1 and 5.5 imply that no new user can obstruct the advancement of user  $i$  if  $i$  is aligned and the immediately preceding position of every  $i_k$  is unoccupied. This is because (1) new users cannot select their initial positions to be immediately preceding any occupied position, and (2) while a new user selecting a position in some queue  $k$ ,  $\text{rm\_critical}_k$  is true. Thus, by the code,  $rm_k$  receives only *select* messages. Since  $rm_k$  does not receive any *advance* or *release* message while the new user is selecting, other users in queue  $k$  cannot advance.

Let  $P_k(i)$  be the position of  $i_k$  in  $q_k$ .

*Claim 5.6.* For any user  $i$  and any  $k$  and  $k' \in R_i$ , it is always true that  $|P_k(i) - P_{k'}(i)| \leq 1$  while  $i_k$  and  $i_{k'}$  are in  $q_k$  and  $q_{k'}$  respectively.

*Proof.* Every  $i_k$ ,  $k \in R_i$  occupies the same position initially (line 9 in Fig. 4). By the code,  $i$  sends an *advance*( $p$ ) message only after it receives a *dec*( $p$ ) message from  $i_k$  and  $i_{k'}$ . By Claim 5.4, if  $i$  receives  $\text{dec}(p)_{k,i}$  and  $\text{dec}(p)_{k',i}$ ,  $i_k$  and  $i_{k'}$  must be at  $p$ . By the code,  $\text{advance\_one\_position}(p)$  is executed when an *advance*( $p$ ) is received. Since  $\text{advance\_one\_position}(p)$  advances  $i_k$  only one position, the claim follows. ■

For any  $k$ ,  $i$  and  $j$ ,  $i \neq j$ , if  $P_k(i) > P_k(j)$ , then we say that user  $i_k$  *waits for* user  $j_k$ , denoted by  $i_k \rightarrow j_k$ .

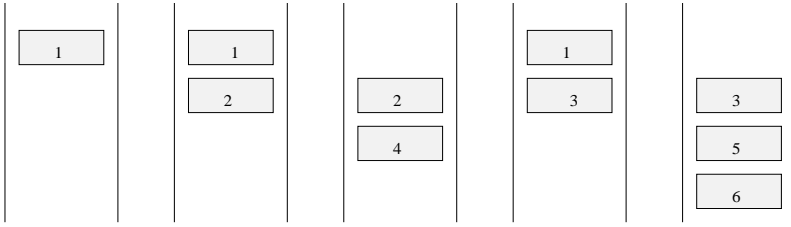


Fig. 6. Two aligned chains of user 1

**Lemma 5.7.** *There is no cycle in the wait-for relation created by the execution of the algorithm.*

*Proof.* We show that if  $i_k \rightarrow j_k$  at time  $t$ ,  $i_{k'} \rightarrow j_{k'}$ , for any resources  $k$  and  $k' \in R_i \cap R_j$ . The lemma is trivially true if  $k = k'$ . Assume  $k \neq k'$ . Without loss of generality, assume that  $i$  selects a position before  $j$  does. If  $j_k$  occupies a position less than  $P_k(i)$ ,  $j_{k'}$  should also occupy a position less than  $P_{k'}(i)$ . This is because when  $j$  selects a position in  $q_k$  and  $q_{k'}$ ,  $P(j_k) = P(j_{k'})$ . By Claim 5.6,  $|P(i_k) - P(i_{k'})| \leq 1$ . Therefore, if  $i_k \rightarrow j_k$ , then  $i_{k'} \rightarrow j_{k'}$ . This is sufficient to show there are no such cycles since a user cannot wait on itself. ■

*Claim 5.8.* Let  $p$  be the lowest position that a user  $i$  is in at some time  $t$ . User  $i$  will be eventually aligned at position  $p$  at some time  $t'$ ,  $t' \geq t$ .

*Proof.* Every  $i_k$ ,  $k \in R_i$  initially occupies the same position. A user  $i$  advances to the next position  $p - 1$  only when it receives an *advance*( $p$ ) message and the message is sent to every  $i_k$ . *advance*( $p$ ) is sent by user  $i$  only when it receives a *dec*( $p$ ) message from every  $i_k$  which is sent only when  $i_k$  occupies  $p$ . Thus, if  $i_k$  is at position  $p$ , then it must have received an *advance*( $p + 1$ ) message. Then every  $i_{k'}$ ,  $k' \neq k$ , will receive it as well and advance to  $p$ .  $i_k$  cannot advance to  $p - 1$  before every  $i_{k'}$  advances to  $p$  because  $i_{k'}$  does not send a *dec*( $p$ ) message until it advances to  $p$ . ■

**Lemma 5.9.** *At some time  $t$ , let  $U_t$  be the set of users at position 1 in every queue. If all users in  $U_t$  leave the critical region and their queues by some time  $t'$ ,  $t' \geq t$ , a user  $i_k$  that is at position  $p$  at  $t$  will eventually advance to  $p - 1$  after  $t$ .*

*Proof.* By Claim 5.8, user  $i$  will be aligned at either  $p$  or  $p - 1$ . If it is at  $p - 1$ , then the lemma is trivially true. Assume that  $i$  is aligned at  $p$ . There are two cases to consider.

(1) Position  $p - 1$  in every queue that  $i$  is in is unoccupied. In this case, this lemma is true by Lemma 5.5.

(2) Position  $p - 1$  in some queue is occupied by some user  $j$ . By Claim 5.8, user  $j$  is (or will be) aligned at position  $p - 1$  or  $p - 2$ . If it is aligned at  $p - 2$ , this lemma is true by Lemma 5.5. Note that no new user can occupy  $p - 1$  left by  $j$ . Assuming that  $j$  is aligned at  $p - 1$ , we can find a chain of users  $x(0), x(1), \dots, x(l)$  for some  $l$ ,  $l \geq 2$ , such that  $x(0)$  is  $i$ , and  $x(m)$ , for some  $m$ ,  $1 \leq m \leq l$ , is aligned at  $p - m$ , and  $x(m)$  and  $x(m + 1)$  are in conflict. Let us call this chain an *aligned chain* of  $i$ . Figure 6 shows an example of two aligned chains of a user 1. The two chains are 1,2,4 and 1,3,5,6.

Without loss of generality, we can assume that there are  $c$  aligned chains of  $i$  for some integer  $c$ ,  $c \geq 1$ .

$$\begin{aligned} &x^0(0), x^0(1), \dots, x^0(l_0) \\ &x^1(0), x^1(1), \dots, x^1(l_1) \\ &x^2(0), x^2(1), \dots, x^2(l_2) \\ &\quad \vdots \\ &x^c(0), x^c(1), \dots, x^c(l_c) \end{aligned}$$

Note that  $x^k(0)$ ,  $1 \leq m \leq c$ , is equal to  $i$ , and for some  $k'$  and  $k''$ ,  $l_{k'}$  and  $l_{k''}$  could be different meaning that each chain can be of different length. Suppose that  $x^k(l_k)$  is at position  $p_{l_k}$ .  $p_{l_k}$  is either 1 or some position higher than 1. (1) If  $x^k(l_k)$  is aligned at 1, then it is in  $U_t$  and will leave its queues at time  $t''$ . (2) If  $p_{l_k}$  is not 1, then by the definition of aligned chain,  $p_{l_k} - 1$  in every queue that  $x^k(l_k)$  is in is unoccupied. Thus, by Lemma 5.5,  $x^k(l_k)$  will advance to  $p_{l_k} - 1$ .

In both cases,  $p_{l_k}$  will be unoccupied. The same is true for every  $l_k$ ,  $1 \leq k \leq c$ . Then, by Lemma 5.5, every  $x^k(l_k - 1)$ ,  $1 \leq k \leq c$ , will advance to  $p_{l_k}$ . Note that new users can occupy  $p_{l_k}$  because  $x^k(l_k - 1)$  occupies  $p_{l_k} - 1$ . By induction on position  $p_{l_k}$ , we can prove that user  $i$  eventually advances to  $p - 1$ . ■

**Theorem 5.10.** *(No-lockout) If a user  $i$  is in the trying region or in the exit region, then it leaves its current region in finite time assuming no user remains in the critical region forever.*

*Proof.* By the no-lockout condition of the subroutine, user  $i$  will eventually select a position in all queues of the resources that it requires. By Lemma 5.7 and by the code, all users at position 1 of queues will receive a *grant* message from all resource managers of the resources they require, and enter the critical region. If they leave the critical region in finite time, then the position 1's occupied by them will be free. Then, as all the users in the position 1 leave their queues, all of position 1's in all queues will be unoccupied eventually. Since all the position 1's in all queues will be unoccupied, by Lemma 5.9, all the users in the queues will advance one position eventually.

Induction on the position that user  $i$  occupies proves that  $i$  eventually leaves the trying region in finite time, and enters the critical region.

Since in the exit region, users send *release* messages and then enter the remainder region immediately, users leave the exit region in finite time. ■

**Theorem 5.11.** *(Concurrency) in any execution of the algorithm, if there is no conflicting user of user  $i$  in the trying region after and when  $i$  enters the trying region, then  $i$  eventually enters the critical region.*

*Proof.* If  $i$  is in the trying region, it will eventually enter the critical region of the subroutine because of the no-lockout condition of the subroutine. Since there is no conflicting user, all  $q_k$ ,  $k \in R_i$ , are empty, user  $i$  will select position 1 in all  $q_k$  and enter the critical region. ■

To measure the response time of the algorithm, we define functions  $Resp_A(c)$  and  $Msg_A$  to be the response time and the message complexity of a resource allocation algorithm  $A$ .

**Theorem 5.12.** *If a resource allocation algorithm  $A$  is used as a subroutine for our algorithm, then our algorithm has response time  $O(\delta c + \delta^2 d + Resp_A(d))$  and message complexity  $O(Msg_A + r\delta)$ .*

*Proof.* For the response time, we only need to show that any user  $i$  that sends a *request* message at time  $t$  will receive *grant* messages from all resource managers  $rm_k$ ,  $k \in R_i(t)$  by time  $t + O(\delta c + \delta^2 d + Resp_A(d))$ .

Note that for a message to be received, it takes time  $d+1$  in the worst case (the maximum message delay plus one step to receive the message). The maximum time period that a user spends in the critical region of the subroutine is  $2d+2$ , i.e., sending a *report* message, receiving a *marked* message (after sending a *select* message, a user leaves the region immediately). Since the response time of the subroutine is  $Resp_A(c)$  and in the subroutine,  $c = 2d+2$ , a user will enter the critical region of the subroutine within time  $Resp_A(2d+2)$ .

Let  $p$  be the initial position of user  $i$ . There are at most  $\delta - 1$  other users in the queues of resources that user  $i$  requires. By Claim 5.6, no user occupies more than two positions at any time. By the code, the position immediately preceding any occupied position is not considered for the selection. Therefore,  $p \leq 3\delta$ .

We now show that it takes at most  $O(pc + p^2 d)$  time for any user  $i$  to leave a queue after it occupies position  $p$ .

It can be shown by an easy induction on  $p$  that if position 1's of all queues are unoccupied at time  $t'$ , then all users at the positions higher than 1 will advance to  $p-1$  within time  $t' + p(2d+2)$ . Assume, by way of induction, that all users at position  $p-1$  advance one position to position  $p-2$  within time  $t' + (p-1)(2d+2)$ . When all users at  $p-1$  advance, all users at position  $p$  send  $dec(p)$  messages, which will be received within time  $d+1$ , and then advance messages will be sent, which takes at most  $d+1$  time to be received. All users at position  $p$  will advance to  $p-1$  by time  $t' + (p-1)(2d+2) + 2d+2$ .

After all  $j_k$ 's,  $k \in R_j$ , occupy position 1 in all  $q_k$ 's, it takes at most  $c + 2d + 2$  time for all  $rm_k$ 's to receive a *release* message from user  $j$ , i.e.,  $d+1$  time for a *grant* message to be received by  $j$ ;  $c$  time for  $j$  to be in the critical region; and  $d+1$  time for a *release* message to be received. Thus, after a user  $i$  occupies position  $p$ , all users at position 1 will leave the queue within time  $c + 2d + 2$ . It follows that within time  $t + Resp(2d+2) + c + 2d + 2 + p(2d+2)$ , every  $i_k$  occupies  $p-1$  or less.

Since there are  $p-1$  positions to advance, within time  $t + Resp(2d+2) + (p-1)(c+2d+2+p(2d+2))$ , user  $i$  advances to position 1. Since  $p \leq 3\delta$ , user  $i$  enters the critical region by time  $t + Resp(2d+2) + (3\delta-1)(c+2d+2+3\delta(2d+2))$  plus

the time to receive a *grant* message ( $d+1$ ). Therefore, user  $i$  enters the critical region by time  $t + O(Resp(d) + \delta c + \delta^2 d)$ .

For the message complexity, each user receives  $r$  *dec* messages and sends  $r$  *advance* messages at each position in the queues. Since a user occupies at most  $3\delta$  positions, the message complexity is  $O(Msg_A + r\delta)$  before it actually uses its required resources in the critical region. ■

One may also wonder about the number of bits that each message requires. Among all the messages in our algorithm, a *marked* message can be largest because it contains information about all the occupied positions in one queue. Since there are at most  $\delta$  conflicting users at a time and all the users in the same queue are conflicting, a *marked* message can contain up to  $O(\delta)$  positions. Because each position can be represented by  $\log P$  bits where  $P$  is the total number of positions in a queue, a *marked* message needs at most  $O(\delta \log P)$  bits.

## 6 Simulation

We are interested in comparing the mean response time of different resource allocation algorithms. We do the comparison by discrete event simulation. This section discusses the simulation results.

We simplify the model in Sect. 2 to reduce the complexity of simulation. We model the execution of the system to be synchronous, so that at every one time unit, each process executes one enabled guarded command (if there is no enabled guarded command, it takes an idle step). However, message passing among processes is still asynchronous so that each message has a random delay with uniform distribution.

The simulation model has 100 users and 100 resources. The resource requirements of processes are set randomly prior to the execution of each simulation, and the conflict graph is constructed based on the resource requirements of all users. We assume a uniform probability distribution of the resource requirements of users (i.e., the probability that a user requires a resource  $a$  is the same as the probability that it requires a resource  $b$ ).

Each user takes idle steps in the remainder region for some random time before it enters the trying region. We call the time period the *thinking time*. The time period that a user uses one resource is called the *resource service time*. The resource service time for each resource is set randomly before a user enters the critical region. The time period that a user stays in the critical region is determined by the sum of the resource service times of all resources that the user needs to use. The probability distributions of the thinking time and the resource service time are also uniform. To increase the accuracy of the simulation, we adjusted simulation run lengths in such a way that the sample mean response times and mean numbers of messages had 95% confidence intervals which were less than 5% of the measured values.

We simulated Chandy and Misra's algorithm (CM), Choy and Singh's algorithm (CS), Awerbuch and Saks' algorithm (AS), our modular algorithm with Choy and Singh's algorithm as a subroutine (CSR) and our modular algorithm with Chandy and Misra's algorithm as a subroutine (CMR). Approximately 300 random conflict graphs were tested.

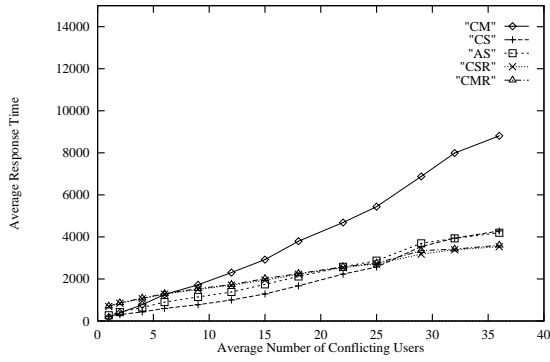


Fig. 7. Average response time when average resource service time = 25 and average message delay = 50

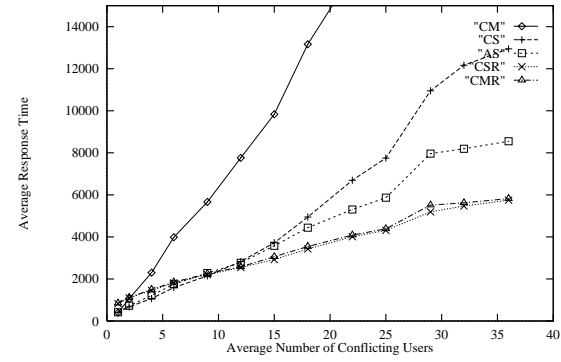


Fig. 10. Average response time when average resource service time = 100 and average message delay = 50

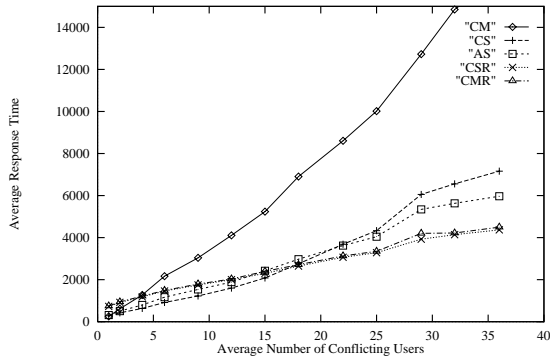


Fig. 8. Average response time when average resource service time = 50 and average message delay = 50

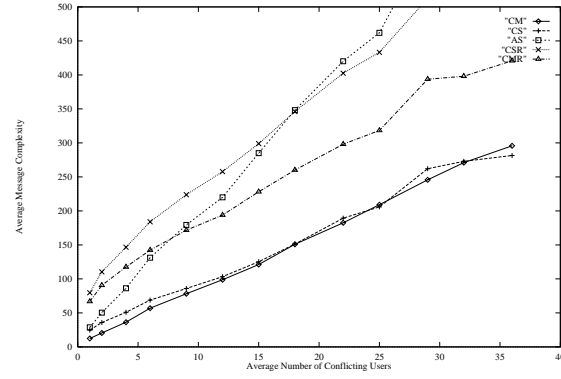


Fig. 11. Average message complexity of when average resource service time = 100 and average message delay = 50

Figures 6, 8, 9, 10 and 11 show the result of our simulations, where the average response times and message complexities are shown for various numbers of conflicting users. To see the impact of the ratio of message delay over the resource service time on the response time, we simulated the algorithms with different ratios. The average message delay was fixed to 50 time units while the resource service time was varied from 25 to 100. The average think time was set to 25.

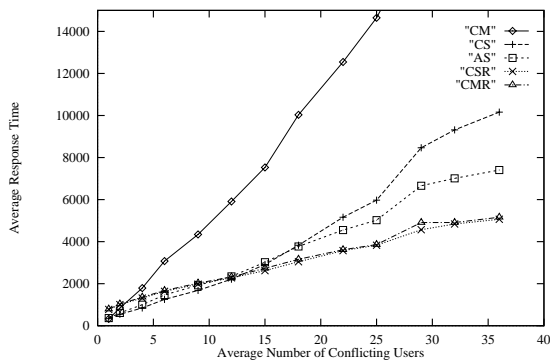


Fig. 9. Average response time when average resource service time = 75 and average message delay = 50

When the ratio of the average response time over the average message delay was small, all the algorithms except CM performed comparably (see Fig. 6). CS performs better than all other algorithms tested when the average number of conflicting users is small. However, as the average number of conflicting users gets bigger, the average response time of CS grows more steeply than those of CSR, CMR and AS. This phenomenon becomes more evident when the ratio of the average resource service time over the average message delay gets bigger (compare Fig. 6 with Figs. 8, 9 and 10).

Our result shows that CSR and CMR give clearly better response time than the other algorithms (AS, CS and CM) when the average number of conflicting users is large and the ratio of the average resource service time over the average message delay is high. CMR and CSR show up to 60% improvement over CS and up to 80% improvement over CM. CMR and CSR show better average performance even than AS (up to 40%) as the ratio gets bigger. CMR works surprisingly better than CM on average. Even in the case where the ratio is small, if the average number of conflicting users is large, CMR and CSR show improvement over all the other algorithms (see Fig. 8). Chandy and Misra's algorithm is worst in every case.

For the message complexity, our combined algorithms naturally require more messages than the subroutines. However, they require fewer messages on average than polynomial response time algorithms such as AS (see Fig. 11).

In summary, the simulation result clearly indicates that the modular algorithm can improve the average response time of its subroutine algorithm with small increase in the messages complexity, especially when there is high contention for resources and the message delay is smaller than the resource service time.

## 7 Conclusion

We presented an efficient modular resource allocation algorithm that uses another resource allocation algorithm of any kind as a subroutine. When our algorithm uses Choy and Singh's dining philosophers algorithm [4], the combined algorithm gives worst case response time  $O(\delta^2(c+d))$  and message complexity  $O(r\delta)$ , which is the fastest known drinking philosophers algorithm with the stated message complexity. When Choy and Singh's dynamic resource allocation algorithm [5] is used, the combined dynamic resource allocation algorithm gives worst case response time  $O(\delta c + (\delta^3 + \log^* |\mathcal{L}|)d)$  and message complexity  $O(r\delta)$ , which is again the fastest known dynamic resource allocation algorithm with the stated message complexity. We also simulated various resource allocation algorithms using a discrete event simulation technique. The simulation results indicate that our algorithm performs better than other algorithms on average, especially when the average number of conflicting users is large and the ratio of the average time period that a user is in the critical region over the average message delay is high.

*Acknowledgments.* The author is greatly indebted to Jennifer Welch for her helpful comments on earlier versions of this paper, and continuous encouragement. This work is supported through her NSF PYI Award CCR-9158478, and IBM Faculty Development Award. Many thanks also go to Ivor Page and Tom Jacob for providing their simulation programs, which helped me understand the algorithm of Awerbuch and Saks. The author would like to thank the anonymous referees for their critical reading of earlier versions of this paper and many helpful comments. Much of this work was done while the author was with the Department of Computer Science, University of North Carolina, Chapel Hill.

## References

1. Awerbuch B, Saks M: A dining philosophers algorithm with polynomial response time, *Proc. 31st IEEE Symposium on Foundations of Computer Science*, St. Louis, MO, pp. 65–74, October 1990,
2. Bar-Ilan J, Peleg D: Distributed resource allocation algorithms, *Proc. 6th International Workshop on Distributed Algorithms*, , pp. 277–291, September 1992
3. Chandy K, Misra J: The drinking philosophers problem, *ACM Transactions on Programming Languages, Systems* 6: 632–646 (1984)
4. Choy M, Singh A: Efficient fault-tolerant algorithms for resource allocation in distributed systems, *Proc. 24th ACM Symposium on Theory of Computing*, pp 593–602, May 1992. Also to appear in *ACM Transactions on Programming Languages, Systems*
5. Choy M, Singh A: Distributed job scheduling using snapshots, *Proc. of the 7th International Workshop on Distributed Algorithms*, , pp. 145–159, September 1993
6. Dijkstra E: hierarchical ordering of sequential processes, *Acta Informatica* 1(2): 115–138 (1971)
7. Lynch N: Upper bounds for static resource allocation in a distributed system, *J Comput Syst Sci* 23: 254–278 (1981)
8. Peterson G, Fisher M: Economical solutions for the critical section problem in a distributed system, *Proc. 9th ACM Symposium on Theory of Computing*, pp 91–97, May 1977
9. Page I, Jacob R, Chern S: Fast algorithms for distributed resource allocation, *IEEE Trans Parallel Distrib Syst*, pp 632–646 (1993)
10. Rhee I: *Efficiency of Partial Synchrony, Resource Allocation in Distributed Systems*, PhD Dissertation, TR94-071, Department of Computer Science, University of North Carolina, Chapel Hill, April 1994 (<ftp://ftp.cs.unc.edu/pub/technical-reports/94-071.ps>. Z)
11. Styer E, Peterson G: Improved algorithms for distributed resource allocation, *Proc. 7th ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, pp 105–116, August 1988.
12. Welch J, Lynch N: A modular drinking philosophers algorithm, *Distrib Comput* 6: 233–244 (1993)
13. Weidman E, Page I, Pervin W: Explicit dynamic exclusion algorithm, *Proc. of the 3rd IEEE Symposium on Parallel, Distributed Processing*, pp 142–149, December 1991

**Injong Rhee** received B.E. in electrical engineering from Kyung-Pook National University, Korea in 1989, and Ph.D. in Computer Science from the University of North Carolina, Chapel Hill, USA in 1994. After conducting postdoctoral research for one year at Warwick University, U.K., and for one year at Emory University, USA, he is now an assistant professor of computer science at North Carolina State University, USA. His research interests include distributed systems and networks, networked multimedia systems, and distributed algorithms.