Distrib. Comput. (2002) 15: 155–175

DISTRIBUTED COMPUTING

The congenial talking philosophers problem in computer networks

Yuh-Jzer Joung*

Department of Information Management, National Taiwan University, Taipei, Taiwan, Republic of China (e-mail: joung@ccms.ntu.edu.tw)

Received: August 2000 / Accepted: November 2001

Summary. Group mutual exclusion occurs naturally in situations where a resource can be shared by processes of the same group, but not by processes of different groups. For example, suppose data is stored in a CD-jukebox. Then when a disc is loaded for access, users that need data on the disc can concurrently access the disc, while users that need data on a different disc have to wait until the current disc is unloaded.

The design issues for group mutual exclusion have been modeled as the *Congenial Talking Philosophers* problem, and solutions for shared-memory models have been proposed [12, 14]. As in ordinary mutual exclusion and many other problems in distributed systems, however, techniques developed for shared memory do not necessary apply to message passing (and vice versa). So in this paper we investigate solutions for Congenial Talking Philosophers in computer networks where processes communicate by asynchronous message passing. We first present a solution that is a straightforward adaptation from Ricart and Agrawala's algorithm for ordinary mutual exclusion. Then we show that the simple modification suffers a severe performance degradation that could cause the system to behave as though only one process of a group can be in the critical section at a time. We then present a more efficient and highly concurrent distributed algorithm for the problem, the first such solution in computer networks.

Key words: Mutual exclusion – Group mutual exclusion – Resource allocation – Distributed algorithms – Message passing

1 Introduction

Group mutual exclusion occurs naturally in situations where a resource can be shared by processes of the same group, but not by processes of different groups. For example, suppose several users are working on a project which has some large data objects stored in a secondary memory such as a CD-jukebox. When a user needs to access a data object, the data object is loaded from the secondary memory to a cache buffer. To increase performance, once a data object is loaded it will remain in the buffer until another data object is requested. So while a data object resides in the buffer, users that need to work on this data object are allowed to access the buffer (possibly a number of times), and users that need a different data object have to wait until no one is working on the data object in the buffer.

To capture the design issues of mutual exclusion intertwined with concurrency, we have presented a problem called *Congenial Talking Philosophers* (CTP) [12]. The problem concerns a set of N philosophers $p_1, p_2, \ldots, p_N$ which spend their time thinking alone and talking in fora. The philosophers may like to hold m different fora $\mathsf{X}_1, \ldots, \mathsf{X}_m$ but, due to the capacity of the meeting room, only one forum can be held at a time. However, more than one philosopher can be in a forum simultaneously. Initially, all philosophers are thinking. From time to time, when a philosopher is tired of thinking, it enters a waiting state where it wishes to attend a forum of its choice. A philosopher can choose only one forum at a time, but it may choose different fora at different waiting states. Given that there is only one meeting room – the shared resource, a philosopher attempting to enter the meeting room to attend a forum can succeed only if the meeting room is empty (and in this case the philosopher starts the forum), or some philosopher interested in the forum is already in the meeting room (and in this case the philosopher joins the ongoing forum). We assume that when a philosopher attends a forum, it spends an unpredictable but finite amount of time in the forum. After a philosopher leaves a forum (that is, exits the meeting room), it returns to thinking.[1] We wish to design an algorithm for the philosophers satisfying the following requirements:

Mutual Exclusion: if some philosopher is in a forum, then no other philosopher can be in a different forum simultaneously.

This research was supported in part by the National Science Council, Taipei, Taiwan, under Grants NSC 86-2213-E-002-053 and NSC 87-2218-E-002-050, and by the 1998 Research Award of College of Management, National Taiwan University.

* Part of the research was done while the author was visiting Lab for Computer Science, Massachusetts Institute of Technology (1999-2000).

[1] Throughout the paper, "in a forum" is used synonymously with "in the meeting room." So, "to attend/leave a forum" is synonymous with "to enter/exit the meeting room."

Lockout Freedom: a philosopher attempting to attend a forum will eventually succeed.
Concurrent Occupancy: if some philosopher p has requested a forum X and no philosopher is currently attending or requesting a different forum, then p can attend X without waiting for other processes to leave the forum (see Appendix A.)

CTP is more general than the ordinary mutual exclusion problem and the Readers and Writers problem [5]. For mutual exclusion, we can dedicate one forum to each philosopher so that only one process can be in the critical section at a time. For the Readers and Writers problem, we can employ a READ operation (forum) for all processes (philosophers) in the system, and a unique WRITE operation for each individual process. A process attempting to read the shared object then requests the READ operation to access the object, while it requests its own WRITE operation when it wishes to update the object. Therefore, a shared object can be concurrently read by different processes, while writing alone must be mutually exclusive.

CTP also bears some similarity to the k-exclusion problem [7,1] in that both allow multiple processes to attend a critical section. However, k-exclusion imposes a bound on the number of processes that may attend the critical section simultaneously, but does not distinguish between processes that may attend concurrently.

We are interested in completely decentralized solutions for the problem. A "semi-distributed" solution can be easily derived, for example, by employing a "concierge" for each forum. A philosopher interested in a particular forum first issues a request to the concierge of the forum. The concierges then compete with one another in a mutually exclusive style to obtain a privilege for their philosophers to use the meeting room. The algorithm is "semi-distributed" because although the contention for the meeting room is resolved in a decentralized manner by the concierges, the decision as to when a set of philosophers interested in the same forum can enter the meeting room is determined in a centralized manner by the concierge of the forum. Furthermore, the "semi-distributed" solution must process each request for a forum in two stages (and thus increases synchronization delay): one between the requesting philosopher and the corresponding concierge, and the other among the concierges for mutual exclusion.

In this paper we focus on CTP in computer networks where philosophers communicate by *reliable* and *FIFO* asynchronous message passing. Solutions for shared-memory models are treated in [12,14,2]. While it is true that shared-memory algorithms can be systematically converted to message passing (or the other way around, see, e.g., Ch. 17 of [16]), such a transformation is generally costly. For example, the transformation of the shared-memory algorithm presented in [12] may result in an asymmetric solution where some processes are designated to maintain the shared variables, usually in a centralized manner, and so the processes are often the bottleneck of the performance. The solution also requires many messages (more than $6(N-1)$) per entry to the critical section. (These phenomena also occur in the transformation of the algorithms in [14,2].) Thus, it is worth investigating solutions that directly take advantage of the underlying features of the execution model. For example, two processes that wish to establish a communication in an asynchronous message-passing system simply execute a pair of **send** and **receive** commands. The system implicitly imposes a causal ordering on the execution so that the **receive** command cannot be executed until the message has arrived. In contrast, a more sophisticated technique is required in a completely decentralized shared-memory model to ensure that two asynchronous processes engaged in a communication are properly synchronized so that the information provider will not overwrite the information before the other process has observed the content.

Indeed, as we shall see shortly in Sect. 3, a symmetric and completely decentralized solution satisfying mutual exclusion, lockout freedom, and concurrent occupancy can be easily devised by slightly modifying Ricart and Agrawala's mutual exclusion algorithm [22].[2] This is not the case we have experienced in the shared-memory model; the algorithm presented in [12] is somewhat complex and is not a straightforward adaptation from existing algorithms for mutual exclusion. Nevertheless, one is easily deceived by this simple algorithm: its behavior appeared to be fine from static analysis until we put it on simulation and learned that it is only slightly better than one imposing mutual exclusion on every entry to the critical section! Therefore, it is also interesting to see why such a simple modification does not work and how a more efficient algorithm can be devised.

The rest of the paper is organized as follows: Sect. 2 presents some metrics for evaluating solutions for CTP. In Sect. 3 we present the straightforward solution described above and analyze why the solution has surprisingly poor performance. Section 4 then presents a more concurrent solution. Conclusions and future work are offered in Sect. 5.

2 Complexity measures

Solutions for CTP can be evaluated by two parameters: *messages* and *time*. For messages, like the ordinary mutual exclusion problem, we are concerned with the number of messages the system generates per entry to the critical section – the meeting room. For the time parameter, we are concerned with how long a philosopher may wait before entering a forum. Instead of using physical time – which would be system dependent and hard to analyze, we use *passages* as the basic metric for evaluating time, as defined below:

Definition 2.1 *A **passage** by p_i through the meeting room is an interval $[t_1, t_2]$, where t_1 is the time p_i enters the meeting room, and t_2 the time it exits the meeting room. The passage is **initiated** at t_1, and is **completed** at t_2. The passage is **ongoing** at any time in between t_1 and t_2. The **attribute** of the passage is $\langle p_i, \mathsf{X}\rangle$, where X is the forum p_i is attending.*

When no confusion is possible, we use intervals (denoted by square brackets $[t_1, t_2]$) and attributes (denoted by angle brackets $\langle p_i, \mathsf{X}\rangle$) interchangeably to represent passages. The

[2] It is interesting to note that Ricart and Agrawala's algorithm can also be straightforwardly extended to solve k-exclusion [19]. As commented earlier, k-exclusion and CTP differ in that in k-exclusion any two processes can be in the critical section simultaneously so long as no more than k processes are in the critical section, while in CTP any number of processes can be in the critical section simultaneously so long as they all belong to the same group. Thus solutions to k-exclusion cannot be directly applied to CTP, and vice versa.

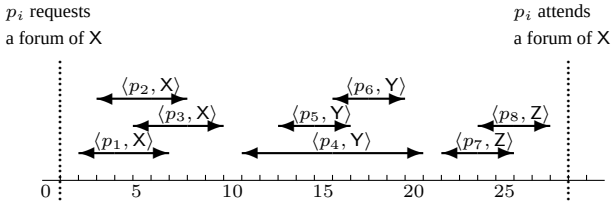

Fig. 1. A layout of passages

phrase "a passage through X by p_i" refers to a passage with attribute $\langle p_i, \mathsf{X}\rangle$.

Definition 2.2 *Let S be a set of intervals. A subset R of S is a **minimal cover** of S if for every $\alpha \in S$, every time instance in α is in some $\beta \in R$ (that is, $\forall\, [t_1, t_2] \in S, \forall\, t_1 \le t \le t_2 : \exists\, [t_3, t_4] \in R, t_3 \le t \le t_4$) and the size of R is minimal. The **dimension** of S, denoted by $\mathbf{dim}(S)$, is the size of a minimal cover of S.*

For example, Fig. 1 shows a scenario in which p_i waits for eight passages before it attends a forum. The five passages $\langle p_1, \mathsf{X}\rangle$, $\langle p_3, \mathsf{X}\rangle$, $\langle p_4, \mathsf{Y}\rangle$, $\langle p_7, \mathsf{Z}\rangle$, and $\langle p_8, \mathsf{Z}\rangle$ constitute a minimal cover of the eight passages shown in the figure. Observe that passages may overlap. So when measuring the "time" p_i waits in terms of passages, we cannot directly count all the passages it waits, but rather their dimension. So in this example only the above five passages suffice to account for p_i's wait.

The ***passage complexity*** is measured by $\mathbf{dim}(T)$, where T is the maximal set of passages that may be initiated after a philosopher p_i has made a request for a forum, and that must be completed before p_i can enter the meeting room. Note that the definition of T does not include those passages that are initiated after p_i has made its request, but p_i need not wait for them to be completed in order to enter the meeting room; that is, we do not count those passages that may be concurrently ongoing with p_i's.

In practice, a "context switch" occurs when a shared resource is "cleaned" for a new group of processes. For example, when a user requires a different data object in the shared objects environment described earlier in Sect. 1, the storage device has to unload the old object and then loads the new one. Depending on the applications, some context switches may be very time-consuming (such as disk loading and unloading). So in CTP a philosopher waiting for more passages through the same forum may in practice need less time than one waiting for fewer passages through different fora. So, in addition to the passage complexity, to measure the waiting time we also need to consider the number of "context switches" a philosopher may wait.

Definition 2.3 *Let U_X be a set of passages through X. Let $t_s = \min\{t \mid [t, t'] \in U_X\}$, and $t_f = \max\{t' \mid [t, t'] \in U_X\}$. Then, U_X is a **round of passages through** X (or simply **a round of** X) if:*

1. *No passage other than those of U_X is initiated in between t_s and t_f; and*
2. *The last passage initiated before t_s and the first passage initiated after t_f, if they exist, are for a different forum.*

In other words, a round of X is a maximal set of consecutive passages through X. If U_X is a round of X, then we say that the round is ***initiated*** at t_s, and ***completed*** at t_f.

The ***forum-switch*** (***context-switch***) ***complexity*** is measured by the maximum number of rounds of passages that may be initiated after a philosopher has made a request for X, but before a round of X is initiated in which p_i makes a passage through X. For example, in Fig. 1 p_i waits for 3 rounds of passages for its request: a round of X, a round of Y, and a round of Z.

3 A straightforward decentralized solution

3.1 The algorithm

Recall that in Ricart and Agrawala's algorithm [22] for n-process mutual exclusion, a process requiring entry to the critical section multicasts a request message to every other process, and enters the critical section only when all other processes have replied to this request. To ensure mutual exclusion and lockout freedom, each process p_i maintains a sequence number SN_i, initialized to 0, that is to be updated according to Lamport's causality rules [15]. To issue a request, a process p_i increases its SN_i by 1, and attaches the pair $\langle i, sn_i\rangle$ to the request, where i is the unique identity of the process and sn_i is the new value of SN_i. $\langle i, sn_i\rangle$ is used as the *priority* of the request.

Upon receiving a request with priority $\langle i, sn_i\rangle$, a process p_j adjusts the value of its SN_j to $\max(SN_j, sn_i)$, and uses the following rules to decide when to reply to the request:

1. p_j replies immediately if it does not require entry to the critical section, or it requires entry to the critical section and the priority of its request is lower than $\langle i, sn_i\rangle$.
2. The reply is delayed if p_j also requires entry to the critical section and the priority of its request is higher than $\langle i, sn_i\rangle$. The reply is delayed until p_j has exited the critical section.

The priority is ordered as follows: a priority $\langle i, sn_i\rangle$ is *higher* than $\langle j, sn_j\rangle$, denoted by $\langle i, sn_i\rangle \prec \langle j, sn_j\rangle$, if and only if $sn_i < sn_j$, or $sn_i = sn_j$ and $i < j$. Also, $\langle i, sn_i\rangle \preceq \langle j, sn_j\rangle$ if and only if $\langle i, sn_i\rangle \prec \langle j, sn_j\rangle$ or $\langle i, sn_i\rangle = \langle j, sn_j\rangle$.

Ricart and Agrawala's algorithm can be straightforwardly modified for CTP as follows: A philosopher wishing to attend a forum X multicasts a request to every other philosopher, and enters the meeting room when all philosophers have replied to its request. A request message is of the form $Req(\langle i, sn_i\rangle, \mathsf{X})$, which additionally bears the name of the forum the philosopher wishes to attend. Also, a reply message (called an *acknowledgment*) is of the form $Ack(j)$, where j is the identity of the replying philosopher.

Upon receipt of the request, a philosopher p_j uses the following rules to decide when to reply to the request:

1. p_j replies immediately if it is not interested in a different forum, or it is interested in a different forum and the priority of its request is lower than that of p_i's request. (A philosopher is *interested* in a forum if it has issued a request for the forum, and either it is still waiting to attend the forum, or it is already in the forum.)
2. The reply is delayed if p_j is interested in a different forum and the priority of its request is higher than that of p_i's request.

```
A.1  *[  wish to attend a forum of X ⟶
A.2        state := waiting;
A.3        SN_i := SN_i + 1;
A.4        multicast Req(⟨i, SN_i⟩, X) to every other philosopher; /* issue a request */
A.5        target := X;
A.6        priority := ⟨i, SN_i⟩; /* store the priority */
B.1  □   receive Req(⟨j, sn⟩, Y) ⟶
B.2        SN_i := max(SN_i, sn); /* adjust sequence number */
B.3        if ⟨j, sn⟩ ≺ priority ⋁ target = Y then
B.4              send Ack(i) to p_j; /* reply to the request immediately */
B.5        else requests_set := requests_set ∪ {Req(⟨j, sn⟩, Y)}; /* defer the reply */
C.1  □   receive Ack(j) ⟶
C.2        acks_set := acks_set ∪ {Ack(j)};
C.3        if |acks_set| = N − 1 then { /* received replies from every other philosopher */
C.4              state := talking; /* enter the meeting room to attend a forum */
C.5              acks_set := ∅; }
D.1  □   exit a forum of target ⟶
D.2        state := thinking;
D.3        target := ⊥;
D.4        priority := ⟨i, ∞⟩; /* reset priority to a minimal value */
D.5        for each Req(⟨j, sn⟩, Y) ∈ requests_set do send Ack(i) to p_j;
D.6        requests_set := ∅;
D.7  ]
```

Variables:

- $state$: the state of p_i; see Fig. 3 for the state transition diagram. The initial state is $thinking$.
- SN_i: p_i's sequence number. It is initialized to 0.
- $priority$: the priority of p_i's request. It is initialized to a minimal value $\langle i, \infty \rangle$.
- $target$: the forum p_i wishes to attend, or $\perp$ otherwise. It is initialized to $\perp$.
- $requests\_set$: the set of requests to which p_i has not yet replied.
- $acks\_set$: the set of replies to p_i's request.

Fig. 2. Algorithm RA1 executed by philosopher p_i

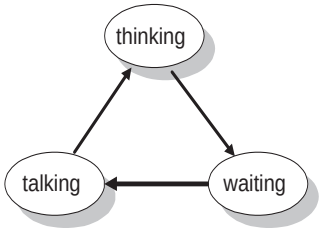

Fig. 3. State transition diagram of a philosopher

We refer to the modified algorithm as RA1. The complete code of RA1 is given in Fig. 2 as a CSP-like repetitive command [11], which is of the form

$$*[gc_1 \;\square\; gc_2 \;\square\; \ldots \square\; gc_k]$$

Each gc_i is called a *guarded command*, which is of the form

$$b; \mathbf{receive}\ msg \longrightarrow S$$

where b is a boolean condition called the *boolean* guard, and "$\mathbf{receive}\ msg$" is called the *message* guard. Both the boolean guard and the message guard are optional. A guarded command can be executed only if it is *enabled*, i.e., its boolean guard evaluates to true and the specified message in the message guard has arrived. The execution receives the message and then the command S is executed atomically. If there is more than one enabled guarded command, then one of them is chosen for execution, and the choice is nondeterministic. We do, however, require that a guarded command that is continuously enabled be executed eventually.

Note that like most algorithms for mutual exclusion (cf. [20]), we have assumed two threads of computation for each philosopher. One is responsible for the philosopher's normal activity in states $thinking$ and $talking$, as well as the transition from $thinking$ to $waiting$ and from $talking$ to $thinking$. The other is responsible for scheduling the transition from $waiting$ to $talking$. It does so by sending and receiving messages to other philosophers, and by making the necessary assignments to local variables. A solution to the (group) mutual exclusion problem then specifies the behavior of the second thread.

Note further that in the algorithm, when p_i issues a request $Req(\langle i, sn \rangle, \mathsf{X})$, we let p_i keep the priority $\langle i, sn \rangle$ in variable $priority$ until it has left X. The priority is reset to a minimal value $\langle i, \infty \rangle$ when p_i is in state $thinking$. As such, a philosopher now possesses a priority all the time, and so the rules for a philosopher to decide when to reply to a request can be simplified as that shown in lines B.3-B.5.

3.2 Analysis of RA1

In this section we prove the correctness of RA1 and analyze its complexity.

Theorem 3.1 *RA1 guarantees mutual exclusion.*

Proof. Assume that p_i and p_j both have issued a request $Req(\langle i,\ sn_i\rangle, \mathsf{X})$ and $Req(\langle j, sn_j\rangle, \mathsf{Y})$, respectively. Without loss of generality, assume that p_i's priority is higher. By the way sequence numbers are maintained, p_j's request must be received by p_i after p_i has issued its request. When p_j's request arrives at p_i, if $\mathsf{X} \neq \mathsf{Y}$, then the reply to the request will be deferred by p_i until p_i has left X. So the two philosophers cannot be in different fora simultaneously. □

For the following theorem, we assume that a request for a forum *ceases to exist* once the requesting philosopher has completed a passage through the forum.

Theorem 3.2 *RA1 guarantees lockout freedom.*

Proof. Observe that the sequence number SN_i maintained by each p_i is nondecreasing, and increases when the philosopher makes an entry to a forum. So after p_i has issued a request $Req(\langle i, sn\rangle, \mathsf{X})$, the number of requests with priorities higher than $\langle i, sn\rangle$ that could occur in the system is bounded. Moreover, because a philosopher can defer the reply to a request only if it has a request with priority higher than that of the requesting philosopher, and because a philosopher spends only finite time in a forum, the requests with priorities higher than $\langle i, sn\rangle$ will eventually cease to exist. Then all other philosophers will reply to p_i's request, thereby ensuring lockout freedom. □

Theorem 3.3 *RA1 allows concurrent occupancy.*

Proof. This follows from the fact that philosophers interested in the same forum do not delay one another's request. □

The following theorem summarizes the static behavior of RA1.

Theorem 3.4 *The message complexity, passage complexity, and forum-switch complexity of RA1 are all* $2(N-1)$.

Proof. For the message complexity, each entry to the meeting room requires $N-1$ requests and $N-1$ replies.

For the passage complexity, let T be the set of passages that are initiated after p_i has issued a request (say $Req(\langle i, sn\rangle, \mathsf{X})$), and that must be completed before p_i can enter the meeting room. We first argue that in between the time (call this t_1) p_i issues a request and the time (call this t_2) it receives replies from all other philosophers, each p_j can have at most two requests with priority higher than that of p_i's request. This is because messages over a communication link are delivered in FIFO order. So if p_i receives a request by p_j after p_i has issued $Req(\langle i, sn\rangle, \mathsf{X})$, then before p_j issues a new request, it must have received p_i's reply to its previous request, and so it must have received p_i's request $Req(\langle i, sn\rangle, \mathsf{X})$ sent before the reply. So p_j's new request must have priority lower than that of p_i's request. So in between t_1 and t_2, p_j can have at most two requests with priority higher than that of p_i's request: one that p_i receives before t_1, and the other that p_i receives after t_1.[3]

[3] The FIFO assumption can be lifted if acknowledgments are also "timestamped" by sequence numbers.

The above argument implies that $|T| \leq 2(N-1)$. It is easy to see that the passages in T can be pairwise disjoint. So the passage complexity of RA1 is $2(N-1)$. The forum-switch complexity is also $2(N-1)$ because two consecutive passages in T can be for different fora. □

We now study how RA1 responds to contention. We first observe that RA1 adopts the following "entry policy":

> *No philosopher can attend a forum if there is a higher priority request from a different philosopher waiting to enter a different forum.*

So the order of entries to the meeting room will entirely be based on request priorities. To analyze RA1, we consider a scenario in which $l \cdot k$ (out of the N) philosophers wish to attend a forum simultaneously. Assume that the philosophers are divided into l groups, each of which consists of k philosophers, such that philosophers in the same group wish to attend the same forum.

Suppose that the priorities of the philosophers' requests are ordered decreasingly as follows:

$$\underbrace{p_1, p_2, \ldots, p_{n_1}}, \underbrace{p_{n_1+1}, p_{n_1+2}, \ldots, p_{n_2}}, \cdots \cdots \underbrace{p_{n_{s-1}+1}, p_{n_{s-1}+2}, \ldots, p_{n_s}}$$

where philosophers $p_{n_{i-1}+1}, p_{n_{i-1}+2}, \ldots, p_{n_i}$ all wish to attend the same forum, and the forum is different from the forum targeted by $p_{n_i+1}, p_{n_i+2}, \ldots, p_{n_{i+1}}$, $1 \leq i \leq s$ (and let $n_0 = 0$). By the algorithm, the $l \cdot k$ requests will result in s rounds of passages (because the philosophers will enter the meeting room in the order of the priorities of their requests), where the ith round consists of $n_i - n_{i-1}$ passages.

Consider the expected value of s and $n_i - n_{i-1}$. Observe that there are $(l \cdot k)!$ possible ways of ordering the $l \cdot k$ requests. Suppose that the $l \cdot k$ requests have equal probability to assume each ordering. Let $E(l,k)$ be the expected number of rounds of passages the requests may generate. In combinatorics, $E(l,k)$ is equivalent to the expected number of runs $l \cdot k$ balls, k balls per color, may generate when they are randomly placed on a line, where a *run* is defined to be a maximal sequence of balls of the same color. By a combinatorial analysis [6], we have

$$E(l,k) = lk - k + 1 \tag{1}$$

Given that there are $l \cdot k$ requests, the expected number of passages per round is

$$\frac{lk}{lk-k+1} < \frac{lk}{lk-k} = \frac{l}{l-1} \tag{2}$$

Therefore, when l increases, the expected number of passages per round tends to be one. So we can expect that when the number of fora m that philosophers may compete for is large, RA1 behaves like a mutual exclusion algorithm where only one philosopher can be in the critical section at a time. Even when there are only two fora (i.e., $m = 2$), at most two philosophers are expected to attend a forum concurrently, regardless of how many philosophers are in each group.

Note that the above analysis is based on the assumption that at any snapshot of the system, the order of priorities of existing requests to the meeting room is randomly distributed. This

assumption in turn is based on the observation that after two philosophers p_i and p_j have attended a forum, the next time they make a request, either one of them can obtain a priority higher than the other, regardless of whose priority was higher in their previous requests. This is because if p_i's new request arrives at p_j before p_j makes its request, then p_j's priority will be lower than p_i's. Similarly, p_i's priority will be lower than p_j's if p_j's new request arrives at p_i before p_i makes its request. If both philosophers make their requests concurrently, then their priorities are determined by their sequence numbers SN_i and SN_j at the time they make the requests, with tie breakers being based on philosopher ids. For simplicity, in the analysis we have assumed an equal chance for either p_i or p_j to win the priority. As we shall see shortly, the results measured in our simulation are close to the above analysis.

3.3 Simulation results

In the simulation, we set up a system of N philosophers and m fora. The forum a philosopher may choose when it wishes to attend a forum is set up in two different ways: *fixed* and *random*. In the fixed forum choice, the N philosophers are divided into m disjoint groups. Each group is assigned a unique forum so that philosophers in the group always choose the forum to attend when they wish to attend a forum. In the random forum choice, a philosopher randomly chooses one of the m fora to attend each time it wishes to attend a forum. To distinguish between these two settings, we put an extra star '$*$' on the value of m to indicate that the forum choice is fixed. Thus, when $m = N*$, the philosophers will use the meeting room in a mutually exclusive style.

The time a philosopher stays in state $thinking$ and in state $talking$ follows an exponential distribution with means $\mu_{thinking}$ and $\mu_{talking}$ respectively. The message transmission time also follows an exponential distribution with a mean μ_{link_delay}. In the simulation we fixed $\mu_{talking}$ to be 250ms, and μ_{link_delay} to be 2ms. The choice of these values came from the observations that CD-ROM access time ranges from 500ms to several milliseconds (depending on the speed)[4], and that message delays in LAN are typically in the order of several milliseconds. Note that we do not consider forum switch cost. If needed, the cost per request can be calculated by multiplying the time needed for a forum switch, and the average number of forum switches a request has to wait. The simulation program is written in Java.[5].

In the first experiment we studied how RA1 reacts to contention. As can be seen, when $\mu_{talking}$ is fixed, the larger the value $\frac{\mu_{talking}}{\mu_{thinking}+\mu_{talking}}$, the more often a philosopher wishes to attend a forum, and so the higher the level of contention. So we used $\frac{\mu_{talking}}{\mu_{thinking}+\mu_{talking}}$ as a metric for contention. We varied $\mu_{thinking}$ from 9000ms to 0ms along the x-axis. We measured the following three values: (i) average *round size*, i.e., the average number of passages per round, (ii) average number of forum switches a request has to wait, and (iii) *throughput ratio* T/T_0, where T is the number of entries to the meeting room per second, and T_0 is the number of entries to the meeting room per second when the philosophers use the meeting room in a mutually exclusive style. For comparison, we set m to the following values $2, 3, 5, 10$, and 30 (for random forum choice), and $2*, 3*, 5*, 10*$, and $30*$ (for fixed forum choice). The results are shown in the six charts of Fig. 4. Each data point in the charts is measured by letting each philosopher make approximately 500 requests.

The two charts on the top show the results measured for average round size. We can see that when the forum choice is random (left chart), the average round size is close to $\frac{m}{m-1}$ as analyzed in the previous section (see Equation (2)), regardless of the level of contention. When the forum choice is fixed, the average round size is close to $\frac{m}{m-1}$ when the contention is low, and it decreases slightly when the level of contention increases. Intuitively, when the contention is high, more philosophers wish to attend the same forum simultaneously. So if an algorithm handles concurrency well, then the average round size should increase. From the experimental results we see that RA1 has poor concurrency performance. Likewise, we can expect that the number of forum switches in RA1 will approach to $N-N/m+1$ when the level of contention increases (see Equation (1) in Sect. 3.2). This is confirmed by the experimental results shown in the two middle charts. Similarly, the throughput ratio measured in the two bottom charts show that little concurrency is offered by RA1.

In the second experiment we studied the performance of RA1 with respect to the number of fora m. Because not all values of m can divide N, we use random forum choice in the experiment. Again, we measured the following: (i) average round size, (ii) average number of forum switches, and (iii) throughput ratio. The results are shown on the three charts of Fig. 5. Each chart shows five curves for each of the following values of $\mu_{thinking}$ (in millisecond): 0, 250, 500, 1000, and 4000, corresponding to the following levels of contention: 100%, 50%, 33.3%, 20%, and 5.9%. From the results we see that when m increases, RA1 performs like a mutual exclusion algorithm regardless of the level of contention.

4 A highly concurrent solution

As analyzed above, the poor concurrency of RA1 is due to the entry policy that no philosopher can attend a forum if there is a higher priority request from a different philosopher waiting to enter a different forum. As a result, if two philosophers p_i and p_j are interested in the same forum, but a third philosopher p_k interested in a different forum has obtained a priority in between p_i's and p_j's, then p_i and p_j cannot attend a forum concurrently. Clearly, concurrency can be increased if p_j can join p_i's forum. To do so, we can weaken the entry policy to the following:

> *No philosopher p_j can attend a forum X if there is a higher priority request from some p_k waiting to enter a different forum, unless there is another p_i already in X such that p_i's request has priority higher than p_k's request.*

In the following we present an algorithm to implement the policy.

[4] See PC Magazine CD-ROM Drives Performance Tests, http://www.zdnet.com/pcmag/features/cdrom/cd-test.htm

[5] Some Java applets animating the algorithms for CTP can be found in http://joung.im.ntu.edu.tw/congenial/

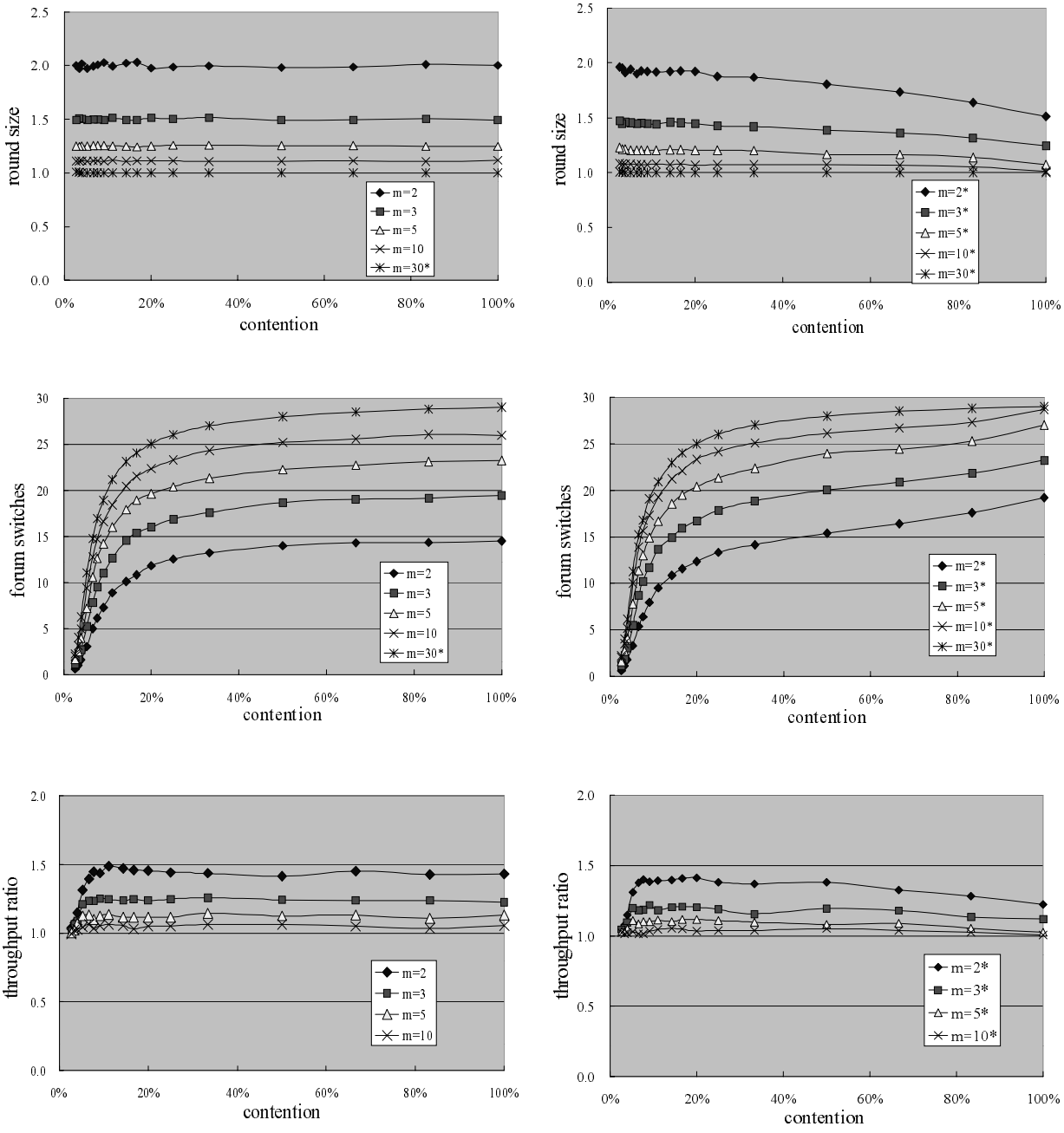

Fig. 4. Performance of RA1 with respect to contention. The left three charts are for random forum choice, where as the right three charts are for fixed forum choice

4.1 The algorithm

By the new entry policy, we can distinguish two types of philosophers in a forum:[6]

- Philosophers that attend the forum because there is no other philosopher with a higher priority such that the philosopher is interested in a different forum. We call these philosophers *captains*.
- Philosophers that attend the forum because there is a captain in the forum. We call these philosophers *successors*.

[6] Note that the definitions of the two types are not disjoint: a philosopher that can potentially become a captain in a forum may become a successor.

In RA1, every philosopher p_i in a forum can be considered as a captain. It enters the forum by receiving acknowledgments to its request from all other philosophers in the system. To implement the concept of successors, we can modify RA1 to let p_i in a forum *capture* other philosophers requesting the same forum to enter the forum. Capturing can be done by leting p_i reply to their requests with a $Start$ message that tells them to enter the meeting room directly without waiting for acknowledgments from other philosophers. For now, we do not allow successors to capture. Section 4.4 considers extensions of the algorithm that allow successors to capture as well.

In the above scenario, when p_i replies a $Start$ to p_j, another p_k with priority lower than p_i may have also issued a request

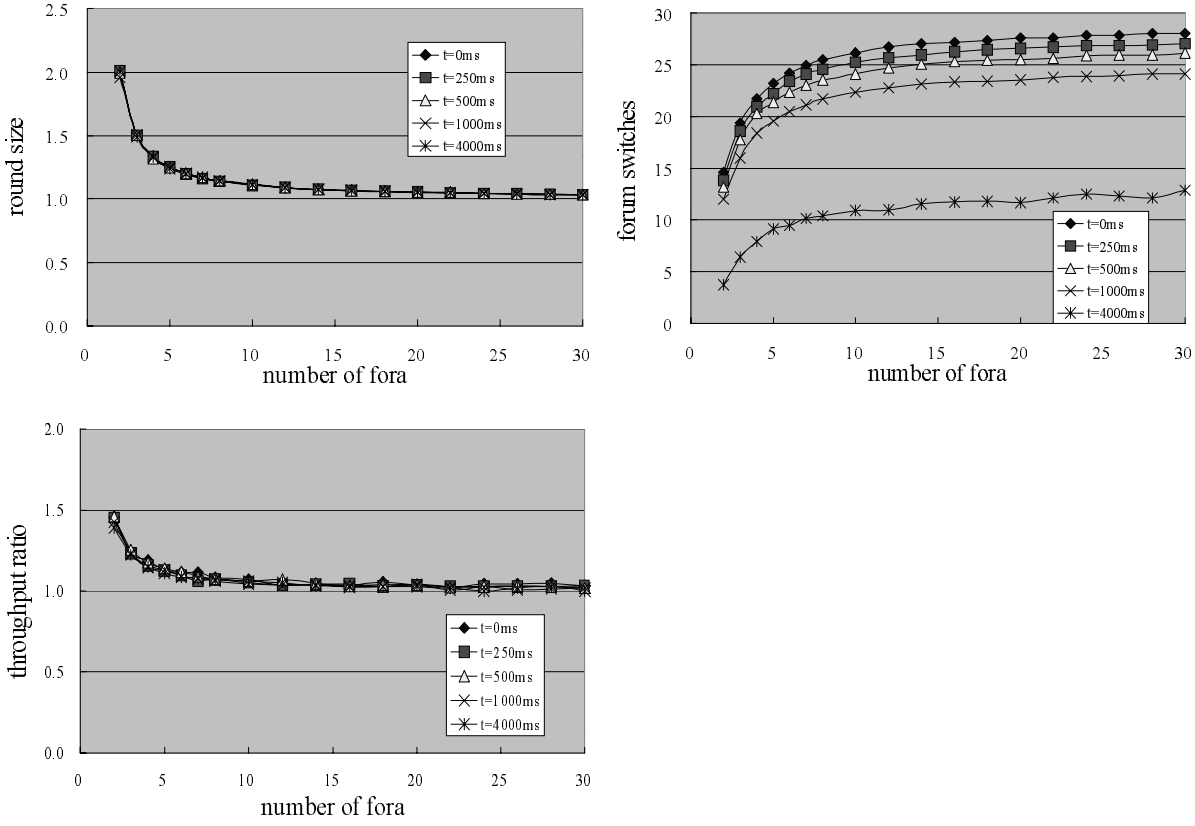

Fig. 5. Performance of RA1 with respect to the number of fora m. Time t in each chart shows the value of $\mu_{thinking}$

for a different forum, and may have already obtained p_j's acknowledgment, waiting for only p_i's reply. So p_j's entry to the meeting room must be known to p_k. To do so, when p_i leaves the forum, in the reply to p_k's request, p_i informs p_k that it has captured p_j to attend a forum. p_k then uses this information to decide if it has an "up-to-date" reply from p_j. If so, then p_k is sure that p_j has also exited the forum and so can enter the meeting room; otherwise, p_k must wait for a new reply from p_j. Clearly, p_j must send such a reply on exiting the meeting room.

From the above discussion we see that sequence numbers alone do not convey enough information for philosophers to decide whether a reply to a request is out-of-date. So in the new algorithm, referred to as RA2, each philosopher p_i maintains a *vector sequence number* VSN_i [18]. VSN_i is a vector of natural numbers of length N and is initialized to contain all zeros. The value $VSN_i[j]$ represents a count of requests by p_j that are known at p_i, either because they originate there (when $j = i$) or because their existence is known through message passing. Let $\mathcal{VSN}$ denote the set of vectors of natural numbers of length N, and $\mathcal{N}$ denote the set of natural numbers. The binary relation '$\prec$' on $\mathcal{N} \times \mathcal{VSN}$ is defined as follows:

$$\langle j, u\rangle \prec \langle k, v\rangle \text{ iff } \textstyle\sum_l v[l] > \sum_l u[l]$$
$$\text{or } (\textstyle\sum_l v[l] = \sum_l u[l] \wedge k > j)$$

It can be seen that the binary relation '$\prec$' on $\mathcal{N} \times \mathcal{VSN}$ is antisymmetric and transitive; and for any two pairs $\langle j, u\rangle$ and $\langle k, v\rangle$ such that $j \neq k$, either $\langle j, u\rangle \prec \langle k, v\rangle$ or $\langle k, v\rangle \prec \langle j, u\rangle$.

Furthermore, p_i also maintains a vector VF_i of natural numbers of length N, where $VF_i[j]$, initialized to 0, represents a count of entries to the meeting room that are made by p_j and that are known at p_i.

The vectors are used as follows: When a philosopher p_i wishes to attend a forum X, it increments $VSN_i[i]$ by 1 and, like RA1, p_i multicasts a request message $Req(\langle i, vsn_i\rangle, \mathsf{X})$ to every other philosopher, where vsn_i is the new value of VSN_i. The value $\langle i, vsn_i\rangle$ is the *priority* of the request. A priority $\langle j, u\rangle$ is *higher than* $\langle k, v\rangle$ if and only if $\langle j, u\rangle \prec \langle k, v\rangle$. Unlike RA1, however, p_i enters the meeting room either because every philosopher has replied with an "up-to-date" acknowledgment Ack, or because some philosopher has replied with a message $Start$. In the former case, p_i enters the meeting room as a captain, (and acts as a captain in the meeting room), while in the latter case p_i enters the meeting room as a successor. Note that many captains may co-exist in a forum.

Upon receiving a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, p_j updates VSN_j to $merge(VSN_j, vsn_i)$, where function $merge(u, v)$ is defined as follows:

$$merge(u, v)[k] = \max(u[k], v[k]), \;\; 1 \leq k \leq N$$

It can be seen that if a request with priority $\langle j, u\rangle$ "happens before" (in Lamport's causality relation [15]) a request with priority $\langle k, v\rangle$, then $\sum_l u[l] < \sum_l v[l]$, and so $\langle j, u\rangle \prec \langle k, v\rangle$.

The rules to decide when and how to reply to the request are as follows:

1. p_j replies with an acknowledgment $Ack(j, vsn_i[i], vf_j)$ (where vf_j is the current value of VF_j) immediately if either (1) it is also interested in X but is not currently acting as a captain in the meeting room, or (2) it is not interested

in X, is not in the meeting room, and has a priority lower than $\langle i, vsn_i \rangle$. (Like RA1, we let p_j possess a priority all the time. The priority is set to $\langle j, vsn_j \rangle$ when p_j issues a request $Req(\langle j, vsn_j \rangle, \mathsf{Y})$, and is reset to a minimal value $\langle j, [\infty, \ldots, \infty] \rangle$ when p_i exits Y.)
2. p_j replies with a message $Start(\langle j, vsn_j \rangle, vsn_i[i])$ if it is in forum X and is acting as a captain in the meeting room. Note that the reply bears p_j's current priority $\langle j, vsn_j \rangle$.
3. Otherwise, p_j must be interested in a different forum, and either p_j is in the meeting room, or its priority is higher than $\langle i, vsn_i \rangle$. In this case, p_j delays the reply until it has exited the meeting room, and then replies to p_i's request with an acknowledgment.

Notice that an acknowledgment $Ack(j, vsn_i[i], vf_j)$ in the algorithm additionally carries two values: $vsn_i[i]$ — p_i's sequence number in its request, and vf_j — p_j's knowledge (at the time the acknowledgment is sent) of the counts of entries to the meeting room made by each philosopher. The value $vsn_i[i]$ is used by p_i to determine whether the acknowledgment is to its current request, or to a previous one. In the latter case the acknowledgment is out-of-date and must be discarded. A philosopher may receive an out-of-date acknowledgment because it can enter the meeting room (as a successor) before it receives all acknowledgments to its request.

The value of vf_j is for p_i to update its VF_i: p_i updates VF_i to $merge(VF_i, vf_j)$ on receipt of $Ack(j, vsn_i[i], vf_j)$. The new value then is used by p_i to check if every acknowledgment $Ack(k, vsn_i[i], vf_k)$ it possesses remains up-to-date. In the algorithm, $VF_i[k]$ must be no greater than $vf_k[k]$ in order for $Ack(k, vsn_i[i], vf_k)$ to remain up-to-date. The intuition is: $VF_k[k]$ must always have a correct count of the number of entries to the meeting room p_k has made. No p_j can have its $VF_j[k]$ greater than $VF_k[k]$ unless some p_l has captured p_k, and has increased $VF_l[k]$ to $VF_k[k] + 1$ before p_k receives p_l's start message. So if $VF_i[k]$ is greater than $vf_k[k]$ carried by p_k's acknowledgment $Ack(k, vsn_i[i], vf_k)$, then p_i knows that p_k has been captured to attend a forum after replying the acknowledgment to p_i. So p_i must wait for a new acknowledgment from p_k to make sure that p_k has exited the forum.

The complete code of the algorithm is given in Fig. 6. Again, the algorithm is presented as a CSP-like repetitive command consisting of five guarded commands A, B, C, D, and E.

Messages are of the following three types:

- $Req(\langle i, vsn_i \rangle, \mathsf{X})$: a request for forum X by p_i, where $\langle i, vsn_i \rangle$ is the priority of the request.
- $Ack(i, sn, vf_i)$: an acknowledgment by p_i to p_j's snth request. vf_i is the value of VF_i at the time the acknowledgment is sent.
- $Start(\langle i, vsn_i \rangle, sn)$: a reply by p_i to p_j's snth request; the reply informs p_j that it can directly enter the meeting room. $\langle i, vsn_i \rangle$ is the priority of p_i at the time the message is sent.

Some detailed comments on the algorithm are provided below. First, consider guarded command B that processes request messages. After p_i has replied with an acknowledgment to $Req(\langle j, vsn_j \rangle, \mathsf{Y})$, p_i may need to send another reply to the request later on. This happens when (1) p_i subsequently enters the meeting room as a captain and wishes to capture p_j to join the forum, or (2) p_i is captured to attend the meeting room and needs to send a new acknowledgment to p_j after it has exited the meeting room. Therefore, a request message may need to be saved for another reply. In the algorithm, request messages are kept in $requests\_set$. Obviously, only the most recent request per philosopher needs to be saved; see lines B.3-4. Lines B.7-13 implement the three rules described earlier regarding how request messages are processed. Each new request is kept in $requests\_set$ (line B.6) until p_i determines that it is out-of-date; see the comment below.

Acknowledgments are processed by guarded command C. p_i enters the meeting room as a captain only when its request has been acknowledged by every other philosopher (line C.10). Then, for each request for X in $requests\_set$, p_i sends a $Start$ message to capture the requesting philosopher to join the forum (lines C.16-19). The requests in $requests\_set$ bearing a priority higher than that of p_i's request can now be deleted (lines C.20-21) because the requesting philosophers must have already entered the meeting room for their requests, for otherwise they will not acknowledge p_i's request. If p_i instead enters the meeting room as a successor (line D.3), then it removes the captain's request and the requests in $requests\_set$ that bear a priority higher than that of the captain's request (lines D.7-8). The captain's priority carried by a start message is used to determine which requests have ceased to exist.

On exiting the meeting room, p_i must reply to the requests that are held in $requests\_set$. Not every request needs to be replied to, though. For example, if the value of $vsn_j[j]$ in a request $Req(\langle j, vsn_j \rangle, \mathsf{Y})$ is less than or equal to $VF_i[j]$, then, obviously, the request has been granted and so no reply is needed. Moreover, if p_i acknowledges a request $Req(\langle j, vsn_j \rangle, \mathsf{Y})$ while it is in Y, then the acknowledgment must carry the most up-to-date count of entries p_i has made to the meeting room. So p_i need not send a new acknowledgment upon exiting the forum. The set $friend\_requests$ is used to store such requests (lines B.14-16 and E.3). Lines E.2-6 determine which request needs to be replied and which needs to be removed.

Note that after p_i has replied to a request (say, by p_j) of the same interest (say, forum Y), p_i may still need to keep the request in $requests\_set$. This is because there is no guarantee that p_j's request will be granted before or concurrently with p_i's request regardless of whose priority is higher. For example, suppose p_i is captured by p_k to attend Y. Suppose further that p_j's request arrives at p_k after p_k has left Y (so that p_k is unable to capture p_j in time), and p_j's request arrives at p_i before p_k has captured p_i. Then, when p_i acknowledges p_j's request, p_j's knowledge of $VF_i[i]$ is one less than the count it learns from p_k's acknowledgment later on. So p_i must send a new acknowledgment to p_j upon exiting Y. Moreover, after p_i has sent the new acknowledgment, p_i may request another forum and then be captured again to enter the meeting room before p_j is allowed to enter the meeting room. So p_i's second acknowledgment again becomes out-of-date and a new one must be issued. In the worst case, p_i needs to keep p_j's request until it learns that the request has ceased to exist.

Finally, suppose p_k captures p_j, and let $Req(\langle j, vsn_j \rangle, \mathsf{X})$ be p_j's request. Because p_k sets its $VF_k[j]$ to $vsn_j[j]$ upon capturing p_j, until p_j receives p_k's start message $VF_k[j]$ is greater than $VF_j[j]$ (whose value is $vsn_j[j] - 1$ when p_j issues its request). In the meantime, another p_i may learn

```
A.1  *[wish to attend a forum of X ⟶
A.2        VSN_i[i] := VSN_i[i] + 1;
A.3        target := X;
A.4        priority := ⟨i, VSN_i⟩; /* obtain a priority for its request */
A.5        state := waiting;
A.6        multicast Req(⟨i, VSN_i⟩, X) to every other philosopher; /* issue a request */

B.1  □ receive Req(⟨j, vsn_j⟩, Y) ⟶
B.2        VSN_i := merge(VSN_i, vsn_j); /* adjust VSN_i */
B.3        if ∃ Req(⟨j, vsn'_j⟩, Z) ∈ requests_set then /* remove p_j's previous request */
B.4             requests_set := requests_set − {Req(⟨j, vsn'_j⟩, Z)};
B.5        if VF_i[j] < vsn_j[j] then {
B.6             requests_set := requests_set ∪ {Req(⟨j, vsn_j⟩, Y)};
B.7             if (target = Y ⋀ ¬is_captain) ⋁ (target ≠ Y ⋀ state ≠ talking ⋀ ⟨j, vsn_j⟩ ≺ priority) then
B.8                  send Ack(i, vsn_j[j], VF_i) to p_j;
B.9             else if target = Y ⋀ is_captain then {
B.10                 send Start(priority, vsn_j[j]) to p_j; /* capture p_j */
B.11                 /* p_i is sure that p_j will make its vsn_j[j] th entry to the meeting room */
B.12                 VF_i[j] := max(VF_i[j], vsn_j[j]);
B.13            } /* otherwise, target ≠ Y ⋀ (state = talking ⋁ priority ≺ ⟨j, vsn_j⟩); the reply is deferred. */
B.14            if target = Y ⋀ state = talking then
B.15                 /* p_i need not re-send a reply to the request on exiting the meeting room * /
B.16                 friend_requests := friend_requests ∪ {Req(⟨j, vsn_j⟩, Y)};
B.17       } /* else the request is out-of-date */

C.1  □ receive Ack(j, sn, vf_j) ⟶
C.2        VF_i := merge(VF_i, vf_j); /* adjust VF_i */
C.3        /* let priority be ⟨i, vsn_i⟩ */
C.4        if vsn_i[i] = sn ⋀ state ≠ talking then { /* otherwise the acknowledgment is out-of-date */
C.5             if ∃ Ack(j, sn', vf'_j) ∈ acks_set then /* remove p_j's previous reply */
C.6                  acks_set := acks_set − {Ack(j, sn', vf'_j)};
C.7             acks_set := acks_set ∪ {Ack(j, sn, vf_j)};
C.8             for each Ack(k, sn'', vf_k) ∈ acks_set do /* remove out-of-date acknowledgments */
C.9                  if vf_k[k] < VF_i[k] then acks_set := acks_set − {Ack(k, sn'', vf_k)};
C.10            if |acks_set| = N − 1 then { /* received acknowledgments from every other philosopher */
C.11                 state := talking; /* enter the meeting room as a captain */
C.12                 VF_i[i] := VF_i[i] + 1;
C.13                 is_captain := true;
C.14                 acks_set := ∅;
C.15                 for each Req(⟨l, vsn_l⟩, Y) ∈ requests_set do
C.16                      if target = Y ∧ VF_i[l] < vsn_l[l] then {
C.17                           /* capture the congenial philosopher */
C.18                           send Start(priority, vsn_l[l]) to p_l;
C.19                           VF_i[l] := max(VF_i[l], vsn_l[l]); }
C.20                      else if ⟨l, vsn_l⟩ ≺ priority then /* the request is out-of-date */
C.21                           requests_set := requests_set − {Req(⟨l, vsn_l⟩, Y)};
C.22            } /* end of if-then statement in line C.10 */
C.23       } /* end of if-then statement in line C.4 */
D.1  □ receive Start(⟨j, vsn_j⟩, sn) ⟶
D.2        /* let priority be ⟨i, vsn_i⟩ */
D.3        if vsn_i[i] = sn ⋀ state ≠ talking then { /* captured by p_i */
D.4             state := talking; /* enter the meeting room as a successor */
D.5             VF_i[i] := VF_i[i] + 1;
D.6             acks_set := ∅;
D.7             for each Req(⟨l, vsn_l⟩, Y) ∈ requests_set do /* remove the captain's request and the
                                  requests that bear a priority higher than that of the captain */
D.8                  if ⟨l, vsn_l⟩ ⪯ ⟨j, vsn_j⟩ then requests_set := requests_set − {Req(⟨l, vsn_l⟩, Y)};
D.9        } /* else, out-of-date message */
```

Fig. 6. Algorithm RA2 executed by philosopher p_i

```
E.1  □ exit a forum of target ⟶
E.2        for each Req(⟨j, vsn_j⟩, Y) ∈ requests_set do
E.3            if VF_i[j] < vsn_j[j] ⋀ Req(⟨j, vsn_j⟩, Y) ∉ friend_requests then
E.4                send Ack(i, vsn_j[j], VF_i) to p_j;
E.5            else if VF_i[j] ≥ vsn_j[j] then /* the request is out-of-date */
E.6                requests_set := requests_set − {Req(⟨j, vsn_j⟩, Y)};
E.7        target := ⊥;
E.8        friend_requests := ∅;
E.9        priority := ⟨i, [∞, . . . , ∞]⟩;
E.10       is_captain := false;
E.11       state := thinking;
E.12 ]
```

Variables:

- $state$: the state of p_i (see Fig. 3 for the state transition diagram). The initial state is $thinking$.
- VSN_i: a vector of length N, where $VSN_i[j]$, initialized to 0, represents a count of requests that are made by p_j and that are known at p_i.
- VF_i: a vector of length N, where $VF_i[j]$, initialized to 0, represents a count of entries to the meeting room that are made by or authorized to p_j and that are known at p_i.
- $priority$: the priority of p_i. It is initialized to a minimal value $\langle i, [\infty, \ldots, \infty]\rangle$.
- $target$: the forum p_i wishes to attend, or $\perp$ otherwise. It is initialized to $\perp$.
- $is\_captain$: a boolean variable indicating if p_i is a captain. It is initialized to false.
- $requests\_set$: set of the most recent requests issued by other philosophers. It is initialized to $\emptyset$.
- $friend\_requests$: set of requests that are for the same forum as p_i's request and that are received while p_i is in the meeting room.
- $acks\_set$: the set of acknowledgments to p_i's request. It is initialized to $\emptyset$

Fig. 6. Algorithm RA2 executed by philosopher p_i (continued)

of the new $VF_k[j]$, and so also has its $VF_i[j]$ greater than $VF_j[j]$. As a result, care must be taken to prevent p_j from miscounting its $VF_j[j]$ using the value learned from other philosophers. In the algorithm, a $Req(\langle j, vsn_j\rangle, \mathsf{X})$ arriving at p_i is considered out-of-date and is removed right away if $VF_i[j] \geq vsn_j[j]$ (line B.5). Therefore, p_j will never receive an $Ack(i, vsn_j[j], vf_i)$ such that $vf_i[j] > VF_j[j]$.

4.2 Analysis of RA2

We now prove the correctness of RA2. For the purpose of analysis, we formalize the terms we used in the algorithm. Recall from the algorithm that for a philosopher p_i to attend a forum X, p_i has to issue a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, and waits until one of the following two conditions is satisfied:

1. it has received an "up-to-date" acknowledgment from every other philosopher (line C.10);
2. it has received a $Start$ from some p_j (line D.3).

In the first case, p_i enters state $talking$ (or, more colloquially, enters the meeting room) as a *captain*, while in the second case p_i enters $talking$ as a *successor*. Also, in the second case, p_j is p_i's captain, and p_j *captures* p_i at the time when it replies the $Start$ message to p_i. In either case, we say that the request has been *granted*. The request *ceases to exist* after p_i has left X. p_i is *waiting for* X in between the time it issues the request until the time the request is granted. p_i is *interested in* X in between the time it issues the request until the time it leaves X.

We begin with mutual exclusion, for which we need the following two lemmas.

Lemma 4.1 *Suppose that p_i and p_j have issued $Req(\langle i, vsn_i\rangle,$ X) and $Req(\langle j, vsn_j\rangle, \mathsf{Y})$, respectively. If $\langle i, vsn_i\rangle \prec \langle j, vsn_j\rangle$, then p_j cannot enter state $talking$ as a captain until p_i has left* X.

Proof. Since p_i's request has a higher priority, the request must be issued before p_j's request arrives, and p_i will not acknowledge p_j's request until p_i has left X. Since in order for p_j to enter state $talking$ as a captain it must receive an acknowledgment from p_i, p_j cannot enter state $talking$ as a captain until p_i has left X. □

Lemma 4.2 *Suppose that a philosopher p_i interested in* X *enters state $talking$ as a captain at time t_1, and leaves the state at t_2. Then, no other philosopher interested in a different forum can enter state $talking$ as a captain until all philosophers captured by p_i in between t_1 and t_2 have left* X.

Proof. Let $Req(\langle i, vsn_i\rangle, \mathsf{X})$ be p_i's request. Suppose that some p_j issuing $Req(\langle j, vsn_j\rangle, \mathsf{Y})$ enters state $talking$ as a captain at time t for some $t \geq t_1$ and $\mathsf{X} \neq \mathsf{Y}$. By Lemma 4.1, $\langle i, vsn_i\rangle \prec \langle j, vsn_j\rangle$. Let p_k be a philosopher that is captured by p_i in between t_1 and t_2, and let $Req(\langle k, vsn_k\rangle, \mathsf{X})$ be p_k's request.

Observe that because the priority of p_i's request is higher than that of p_j's request, p_j cannot receive p_i's acknowledgment to $Req(\langle j, vsn_j\rangle, \mathsf{Y})$ until p_i has left X. Let $Ack(i, vsn_j[j], vf_i)$ be the acknowledgment. Since p_i captures p_k, $vf_i[k] \geq vsn_k[k]$. So, immediately before p_j enters state $talking$, $VF_j[k] \geq vf_i[k] \geq vsn_k[k]$ (because p_j has updated its $VF_j[k]$ via the value of $vf_i[k]$ carried by p_i's acknowledgment). In order for p_j to enter state $talking$ as a captain the acknowledgment from p_k, say $Ack(k, vsn_j[j], vf_k)$, must be up-to-date. That is, $vf_k[k] \geq VF_j[k]$. So $vf_k[k] \geq vsn_k[k]$.

To send out such an acknowledgment, p_k must have left X, for otherwise p_k cannot increment its $VF_k[k]$ to $vsn_k[k]$. So p_j cannot enter state $talking$ as a captain while p_k is still in X. □

Theorem 4.3 *RA2 guarantees mutual exclusion.*

Proof. Suppose by contradiction that p_i is in state $talking$ attending X and, at the same time, p_j is in state $talking$ attending Y, where X ≠ Y. There are four cases to consider, depending on whether they enter state $talking$ as a captain or as a successor. By Lemma 4.1, p_i and p_j cannot both enter the state as a captain; and by Lemma 4.2, it is not possible, either, that one of them enters state $talking$ as a captain while the other as a successor. Moreover, since a captain can capture philosophers only when it is in state $talking$, by Lemma 4.2, p_i and p_j cannot both enter state $talking$ as a successor. Contradiction. □

We now prove lockout freedom, for which we need the following lemmas.

Lemma 4.4 *While p_i is waiting for X and before it has left X, every other p_j can make at most two entries to the meeting room as a captain to attend a different forum.*

Proof. To enter the meeting room as a captain, a philosopher must receive acknowledgments to its request from every other philosopher. Due to the vector sequence numbers maintained by the philosophers and the FIFO delivery in each communication link, after p_i has issued a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, each p_j can issue at most one request with a priority higher than $\langle i, vsn_i\rangle$ if p_j subsequently enters the meeting room as a captain. This is because p_i's acknowledgment to p_j's request must be received by p_j after $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has arrived. So subsequent requests by p_j must have a priority lower than $\langle i, vsn_i\rangle$. While waiting for X and before leaving X, p_i does not acknowledge any request for a different forum with a lower priority. So each p_j interested in a different forum can make at most one entry to the meeting room as a captain in such a way that the request is received by p_i after p_i has issued $Req(\langle i, vsn_i\rangle, \mathsf{X})$.

Clearly, while p_i is waiting for X, p_j can make at most one entry to the meeting room to attend a different forum in such a way that the request is received by p_i before p_i issues $Req(\langle i, vsn_i\rangle, \mathsf{X})$. Overall, after p_i has issued $Req(\langle i, vsn_i\rangle, \mathsf{X})$ and before it has left X, p_j can make at most two entries to the meeting room as a captain to attend a different forum. □

The above lemma implies that after p_i has issued a request for X, at most $2(N-1)$ entries to the meeting room can "overtake" p_i's request in such a way that each entry is for a forum different from X, and the philosopher making this entry acts as a captain in the meeting room.

Lemma 4.5 *Assume $j \neq i$. Then, at any time instance, either (1) $VF_j[i] \leq VF_i[i]$, or (2) $VF_j[i] = VF_i[i]+1$, p_i has made a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, where $vsn_i[i] = VF_i[i]+1$, some philosopher p_k has replied to the request with a start message, and p_i has not yet received the message.*

Proof. Recall from the algorithm that p_j changes $VF_j[i]$ only when (a) it receives an $Ack(l, sn_l, vf_l)$ such that $vf_l[i] > VF_j[i]$ (line C.2), or (b) it is in the meeting room and has replied to a request $Req(\langle i, vsn'_i\rangle, \mathsf{X}')$ with a start message (lines B.12 and C.19). Since $VF_j[i] = VF_i[i] = 0$ initially and since $VF_i[i]$ is incremented only when p_i has made an entry to the meeting room, if no philosopher has ever replied to p_i's requests with a start message, then $VF_j[i] \leq VF_i[i]$ all the time.

So if $VF_j[i] = v_1 > VF_i[i] = v_2$ at some time t, then there must exist some p_k such that p_k is the first philosopher to set its $VF_k[i]$ to a value (say v_3) greater than v_2. Let $t' \leq t$ be the time p_k sets $VF_k[i]$ to v_3. Then, p_k must have received a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ by p_i such that $vsn_i[i] = v_3$, have replied to the request with a start message, and have set $VF_k[i]$ to v_3 via lines B.12 or C.19. Clearly, at this moment p_i must have not yet made its v_3th entry to the meeting room; otherwise, $VF_i[i]$ would have a value no less than v_3, contradicting the assumption that $VF_i[i]$ is still less than v_3 at time t. So $VF_i[i] < v_3$ at time t'. Since p_i has already made its v_3th request, $VF_i[i] \geq v_3 - 1$ at t'. So $VF_i[i] = v_3 - 1$ at t'.

Let t'' be the time p_i advances $VF_i[i]$ to v_3. Observe that p_i will not issue any new request in between t' and t''. So for every philosopher p_l, $l \neq i$, p_l can advance $VF_l[i]$ to at most v_3, regardless of how it learns of this value. Moreover, p_i must not have received p_k's start message in between t' and t'', for, otherwise, p_i will make its v_3th entry to the meeting room. The fact that $VF_i[i] = v_3 - 1$ at t' and $t' \leq t$ implies that $VF_i[i] \geq v_3 - 1$ at t. However, $VF_i[i] = v_2 < v_3$ at t. So $VF_i[i] = v_3 - 1$ at t, and $t \leq t''$. So $VF_j[i]$ can be advanced to at most v_3 at t. Since at time t $VF_j[i] = v_1 > VF_i[i] = v_3 - 1$, $VF_j[i] = v_3$ at t.

So at time t, $VF_j[i] = VF_i[i] + 1$, p_i has made a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ where $vsn_i[i] = VF_i[i] + 1$, some philosopher p_k has replied to the request with a start message, and p_i has not yet received the message. The lemma therefore follows. □

Lemma 4.6 *Suppose p_j has received a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ by p_i. Then, when the request is removed from p_j's $requests\_set$, either (1) the request has ceased to exist, or (2) p_i is already in the meeting room attending X, or some philosopher p_k has replied to the request with a start message (but the message has not yet arrived at p_i).*

Proof. When p_j receives a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, it keeps the request in $requests\_set$ until (a) p_j receives another request from p_i (line B.4), (b) p_j enters the meeting room as a captain to attend a different forum, and p_j's priority is lower than $\langle i, vsn_i\rangle$ (line C.21), (c) $VF_j[i]$ is greater than or equal to $vsn_i[i]$ (lines B.17 and E.6), or (d) p_j enters the meeting room as a successor and the priority of p_j's captain is lower than or equal to $\langle i, vsn_i\rangle$ (line D.8).

In Case (a), since messages are delivered in FIFO order, when p_j receives another request from p_i, p_i must have already attended X and so the request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has ceased to exist.

In Case (b), by Lemma 4.1, when p_j enters the meeting room as a captain, p_i must have attended X and so the request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has ceased to exist.

In Case (c), $VF_i[i]$ must be equal to $vsn_i[i] - 1$ when p_i makes the request $Req(\langle i, vsn_i\rangle, \mathsf{X})$, and remains in the same value until p_i attends X (note that p_i will never advance $VF_i[i]$ using a value learned from another philosopher). By

Lemma 4.5, when p_j finds that $VF_j[i] \geq vsn_i[i]$, it must be the case that either p_i's request has ceased to exist, or some philosopher has replied to the request with a start message. So the lemma is also proven for this case.

In Case (d), let p_k be p_j's captain, and let $\langle k, vsn_k \rangle$ be p_k's priority. Clearly, if $k = i$ then we are done. So assume that $i \neq k$. Since p_k enters the meeting room as a captain, and since p_i's priority is higher than p_k's, p_k must have already received p_i's request before p_k enters the meeting room. There are two subcases to consider.

1. p_i's request is still in p_k's $requests\_set$ before p_k enters the meeting room. Then p_k and p_i must be interested in the same forum. Then p_k will also reply to p_i's request with a start message when p_k enters the meeting room. So we are done.
2. p_i's request has already been removed from p_k's $requests\_set$ when p_k enters the meeting room. Then p_k must have removed the request because of a reason similar to one of the above four cases. In any case, it is easy to see that either p_i has already attended X, or some philosopher has sent p_i a start message. So the case is also proven. □

Lemma 4.7 *Suppose p_i's request $Req(\langle i, vsn_i \rangle, \mathsf{X})$ has the highest priority among all existing requests. Then, p_i will eventually enter the meeting room.*

Proof. Given that every message is eventually delivered, we may assume that no message is currently in transit. So every philosopher has received p_i's request. Moreover, we shall assume that every philosopher still keeps the request in its $requests\_set$. Otherwise, the lemma is proven because by Lemma 4.6 p_i must have entered, or will eventually enter, the meeting room.

When a philosopher p_j receives p_i's request, either (1) it replies to the request immediately, or (2) it defers the reply until it has left a forum (lines B.7-13). In the latter case, p_j must be interested in a forum different from X, and either (2.1) p_j is already in the meeting room, or (2.2) p_j has not yet entered the meeting room, but its request has priority higher than that of p_i's request. In Case (2.1), p_j will eventually reply to p_i's request when it exits the meeting room (lines E.2-4) because it can spend only finite time in the meeting room. Case 2.2 cannot be applied here because by the lemma assumption p_i's request has the highest priority in the system. So in either Case (1) or Case (2), p_j will reply to p_i's request.

So assume that every p_j, $j \neq i$, has replied to p_i's request. If some of the replies is a $Start$, then clearly p_i will enter the meeting room. Otherwise, all the replies are acknowledgments. If every acknowledgment $Ack(j, vsn_i[i], vf_j)$ is up-to-date, i.e., $VF_i[j] \leq vf_j[j]$, then p_i will enter the meeting room (lines C.8-10). So for the rest of the proof it remains to consider the case that some $Ack(j, vsn_i[i], vf_j)$ is out-of-date.

We argue that if $Ack(j, vsn_i[i], vf_j)$ is out-of-date, i.e., $VF_i[j] > vf_j[j]$, then p_j will send p_i a new acknowledgment $Ack(j, vsn_i[i], vf'_j)$ with $vf'_j[j] = vf_j[j] + 1$. To see this, observe that the value of $VF_j[j]$ must be equal to $vf_j[j]$ when p_j sent $Ack(j, vsn_i[i], vf_j)$. So when p_i considers the acknowledgment out-of-date, either $VF_j[j] = vf_j[j]$, or $VF_j[j] > vf_j[j]$. If $VF_j[j] = vf_j[j]$, by Lemma 4.5 p_j will make an entry to the meeting room. So p_j will send a new acknowledgment $Ack(j, vsn_i[i], vf'_j)$ with $vf'_j[j] = vf_j[j] + 1$ to p_i after exiting the meeting room. If $VF_j[j] > vf_j[j]$, then p_j must have made an entry to the meeting room after sending $Ack(j, vsn_i[i], vf_j)$ to p_i. So p_j will re-send p_i a new acknowledgment $Ack(j, vsn_i[i], vf'_j)$ with $vf'_j[j] = vf_j[j] + 1$.

If p_j's new Ack is up-to-date, then p_i eventually enters the meeting room. Otherwise, p_j must re-enter the meeting room. However, since p_i's request has the highest priority, by Lemma 4.1, no philosopher can act as a captain attending a different forum while p_i is waiting for X. So p_j can only re-enter the meeting room to attend X. If p_j re-enters the meeting room as a captain, then since p_j still keeps p_i's request in its $requests\_set$, p_j will reply to p_i's request with a $Start$ message when it enters the meeting room (lines C.15-19). If p_j re-enters the meeting room as a successor, then there must exist some p_k which acts as a captain in the meeting room attending X. By our assumption, p_k has also received p_i's request. If p_k received p_i's request before it entered the meeting room, then p_k must have sent a $Start$ to p_i when it entered the meeting room (lines C.15-19), regardless of the fact that it has already replied an Ack to p_i. If p_k received p_i's request after it has entered the meeting room, then p_k must have also replied a $Start$ to p_i upon receiving p_i's request (lines B.9-12). In either case, p_i will eventually receive p_k's $Start$ message and will then attend X. □

Theorem 4.8 *RA2 guarantees lockout freedom.*

Proof. From Lemma 4.4, after a philosopher p_i has issued a request for X, at most $2(N-1)$ entries to the meeting room can precede p_i's such that each of the entries is for a forum different from X, and the philosopher making this entry acts as a captain in the meeting room. Because of the following three facts:

- only a philosopher acting as a captain in the meeting room can capture philosophers to enter the meeting room,
- a philosopher spends only finite time in the meeting room, and
- Lemma 4.7 that the request with the highest priority will eventually be granted,

after p_i has issued a request for X, it will eventually enter the meeting room to attend X. □

Theorem 4.9 *RA2 allows concurrent occupancy.*

Proof. This follows directly from the fact that if some philosophers have issued requests for X and no philosopher is interested in a different forum, then all the requests for X will be acknowledged immediately when they are received by their destination philosophers. So the philosophers interested in X can attend the forum concurrently. □

4.3 Performance analysis

In this section we analyze the performance of the algorithm from various perspectives. For message complexity, because some responses to a request may be caused by other requests, we measure the amortized cost.

Theorem 4.10 (Message Complexity) *The amortized message cost per entry to the meeting room is at most $3(N-1)$.*

Proof. A philosopher p_i multicasts a request $Req(\langle i, vsn_i\rangle, \mathsf{X})$ to every other philosopher when it wishes to attend a forum. A philosopher p_j responds to p_i's request in one of the following ways:

1. It does not reply any message (because it learns from another philosopher that p_i has been captured to attend a forum).
2. It replies to the request with exactly one message – either a $Start$ or an Ack.
3. It replies to the request with an Ack and then a $Start$.
4. It replies to the request with at least two Acks. There are two subcases: either p_j's replies are all Acks, or are a sequence of Acks followed by a $Start$.

To simplify the proof, in Case 4 we consider only the second subcase. The first subcase can be proved similarly. Let the replies in this case be $msg_1, msg_2, \ldots, msg_{l+1}$. This case happens when p_j, which has also made a request, say, Req_1, is captured to attend a forum after replying msg_1 to p_i. p_j sends msg_2 to p_i on exiting the forum. Then p_j makes a second request Req_2, is captured again to attend a forum, and sends msg_3 to p_i on exiting the forum; and so on until p_j sends msg_l to p_i. Then p_j makes the lth request Req_l, enters the meeting room as a captain, and sends msg_{l+1} (which is a $Start$) to p_i.

We can envisage each msg_k, $1 < k \leq l+1$, as being caused by Req_{k-1} so that msg_k is counted as a "reply" by p_i to Req_{k-1}. So p_j replies to $Req(\langle i, vsn_i\rangle, \mathsf{X})$ with msg_1, while p_i "replies" to Req_{k-1} with msg_k. Still, p_i may actually reply some messages to $Req_1, \ldots, Req_l$. Clearly, if p_i does not reply to any of the requests with an Ack and then a $Start$ (i.e., Case 3 above), then each request message generates at most two replies by the receiver. So the amortized cost per entry to the meeting room is at most $3(N-1)$, and so we are done with the proof. Even if p_i does reply to one of the l requests with an Ack and then a $Start$, we can still amortize the cost to $3(N-1)$ because only one message (i.e., msg_1) is counted as a reply by p_j to $Req(\langle i, vsn_i\rangle, \mathsf{X})$. In the following we show that indeed p_i can reply to at most one of these l requests with an Ack and then a $Start$, thereby establishing the theorem.

We first show that after $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has ceased to exist, p_i cannot make a new request (say, Req'), and reply a $Start$ to Req_k when Req' is granted for any $1 \leq k \leq l$. This is because p_i must receive p_j's Ack to Req' before it can reply a $Start$ to Req_k. This Ack must be sent before msg_{k+1}; otherwise, because messages are delivered in FIFO order, p_i will receive msg_{k+1} before it receives the Ack. Then by msg_{k+1} p_i will know that Req_k has ceased to exist and so will not reply a $Start$ to Req_k. On the other hand, the fact that p_j's Ack to Req' is sent before msg_{k+1} implies that p_j receives Req' before it replies msg_{k+1} to $Req(\langle i, vsn_i\rangle, \mathsf{X})$. By Req' p_i knows that $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has ceased to exist, and so will not reply msg_{k+1} to $Req(\langle i, vsn_i\rangle, \mathsf{X})$; contradiction.

Next, we show that before p_i issues $Req(\langle i, vsn_i\rangle, \mathsf{X})$, p_i cannot make a request (say, Req'') and reply a $Start$ to any Req_k when the request is granted. Clearly, because msg_1 is sent after p_i issues $Req(\langle i, vsn_i\rangle, \mathsf{X})$, Req'' must have ceased to exist when p_j issues Req_2 (and the following requests). So it remains to consider whether or not p_i can reply a $Start$ to Req_1 after Req'' is granted (and before it ceases to exist). If this is case, then p_j must receive Req'' and reply an Ack to the request before it replies msg_1 to $Req(\langle i, vsn_i\rangle, \mathsf{X})$. When p_i receives p_j's Ack and enters the meeting room as a captain, the $Start$ message it replies to Req_1 must arrive at p_j before $Req(\langle i, vsn_i\rangle, \mathsf{X})$ does (because, again, messages are delivered in FIFO order). So p_j must have already attended a forum for Req_1 when it receives $Req(\langle i, vsn_i\rangle, \mathsf{X})$. Recall that msg_2 is sent when p_j exits the forum. So msg_1 is sent while p_j is still in the forum. By the algorithm, if p_j has replied an Ack to $Req(\langle i, vsn_i\rangle, \mathsf{X})$ when it is in a forum, then the request will be placed in $friend\_requests$ (line B.16) and p_j will not reply another message to the request on exiting the forum (line E.3). This then contradicts the fact that p_j replies msg_2 to $Req(\langle i, vsn_i\rangle, \mathsf{X})$ on exiting the forum.

From the above arguments we see that if p_i replies a $Start$ to some Req_k, $1 \leq k \leq l$, then the $Start$ must be sent after $Req(\langle i, vsn_i\rangle, \mathsf{X})$ has been granted, and before the request ceases to exist. So for any subsequent request Req_h, $k < h \leq l$, if p_i replies to Req_h with a $Start$, then p_i must still be in the forum requested by $Req(\langle i, vsn_i\rangle, \mathsf{X})$, and will not reply an Ack to Req_h before it replies the $Start$. So only one message is replied by p_i to Req_h. So p_i can reply to at most one of the requests $Req_1, \ldots, Req_l$ with an Ack and then a $Start$. This completes the proof of the theorem. □

Theorem 4.11 (Forum-Switch Complexity) *After a philosopher p_i has issued a request for X, at most $2(N-1)$ rounds of passages can be initiated before a round of X is initiated in which p_i makes a passage through X.*

Proof. By Lemma 4.2, the passages made by a captain and its successors must belong to the same round. So every round of passages must be initiated by a philosopher that attends the meeting room as a captain. By Lemma 4.4, after p_i has issued a request for X, at most $2(N-1)$ rounds of passages can be initiated before a round of X is initiated in which p_i makes a passage through X. The lemma therefore follows. □

Lemma 4.12 *Suppose a philosopher p_i makes a passage $\alpha = [t_{i,s}, t_{i,f}]$ through the meeting room, and p_i acts as a captain in the meeting room. Suppose further that k philosophers have been captured by p_i while p_i is in the meeting room (where each of them may be captured more than once). Let R be the set of passages consisting of α and the passages made by the k philosophers when they are captured by p_i. Then, $\mathbf{dim}(R) \leq k+1$.*

Proof. Let p_j be a philosopher that has been captured by p_i in α, and let $\beta = [t_{j,s}, t_{j,f}]$ be the passage made by p_j when it is captured. Clearly, $t_{j,s} > t_{i,s}$. Moreover, if p_j has been captured again by p_i after it has made β, then $t_{j,f} < t_{i,f}$; that is, β is contained in α. So, to calculate $\mathbf{dim}(R)$ we can simply calculate $\mathbf{dim}(R')$, where R' consists of α and, for each p_j that has been captured by p_i in α, the last passage γ_j made by p_j when it is captured by p_i in α. Because two different γ_j and γ_l (that are intervals) need not intersect, we have $\mathbf{dim}(R) \leq k+1$. □

For the following theorem, we say that a philosopher is the *pioneer* of a round of passages if the philosopher initiates the first passage of the round (where the philosopher must attend a forum as a captain).

Theorem 4.13 (Time Complexity) *Suppose that a philosopher p_i has made a request. Let R be the set of passages that are initiated after p_i has made the request, and that must be completed before p_i can enter the meeting room. Then,* $\mathbf{dim}(R) \leq (N-1)(3N-2)/2$.

Proof. Let t_1 be the time at which p_i makes its request (say $Request(\langle i, vsn_i \rangle, \mathsf{X})$), and t_2 be the time p_i enters the meeting room to attend X. Suppose that R consists of l rounds of passages $R_1, R_2, \ldots, R_l$. By the mutual exclusion property of the algorithm, $\mathbf{dim}(R) = \sum_{1 \leq k \leq l} \mathbf{dim}(R_k)$.

Clearly, of the l rounds of passages, at most one of them is initiated before t_1. By Lemma 4.4, every other p_j can establish at most two rounds of passages in between t_1 and t_2 such that p_i must wait until the passages in the two rounds are completed before p_i can enter the meeting room. So $l \leq 2(N-1)+1$. However, recall from the proof of Lemma 4.4 that of the 2 rounds of passages p_j establishes, one must be initiated in the way that p_i acknowledges p_j's request before t_1, while the other is initiated in the way that p_i acknowledges p_j's request after t_1. So if a philosopher has initiated a passage α in between t_1 and t_2 such that α belongs to a round that is initiated before t_1, then the philosopher can establish at most one round of passages in between t_1 and t_2. So for the worst case scenario, we may assume that $l = 2(N-1)$ and all the rounds $R_1, R_2, \ldots, R_l$ are initiated after t_1. Let $R_1, R_2, \ldots, R_{N-1}$ be the rounds that are initiated in the way that p_i acknowledges the pioneers' requests before t_1, and $R_N, R_{N+1}, \ldots, R_{2(N-1)}$ are initiated in the way that p_i acknowledges the pioneers' requests after t_1.

Consider each round R_k. Observe that if a philosopher p_j has been captured by the pioneer of R_k while the pioneer is in the meeting room, then p_j's next request must be received by p_i after t_1, and so p_j cannot be the pioneer of $R_1, R_2, \ldots, R_{N-1}$ after it has made the new request. However, p_j can still be the pioneer of $R_N, R_{N+1}, \ldots, R_{2(N-1)}$ because p_j's new request may still have a priority higher than the priority of p_i's request $Request(\langle i, vsn_i \rangle, \mathsf{X})$. This means that for each pioneer of $R_N, R_{N+1}, \ldots, R_{2(N-1)}$, the pioneer of the round may capture at most $N-2$ philosophers; while for each pioneer of $R_1, R_2, \ldots, R_{N-1}$, one can capture no philosopher, one captures one, one captures two, ..., and one captures $N-2$. By Lemma 4.12, therefore,

$$\begin{aligned} \mathbf{dim}(R) &= \sum_{1 \leq k \leq l} \mathbf{dim}(R_k) \\ &= (N-1)(3N-2)/2 \end{aligned}$$

□

4.4 Simulation results

The simulation was conducted in a setting similar to that for RA1 (see Sect. 3.3). In the first experiment we studied how RA2 reacts to contention. We considered five different values of m (2, 3, 5, 10, and 30) for both random forum choice and fixed forum choice. The results are shown in Fig. 7. From the two top charts we see that the average round size increases as the level of contention increases. The two second charts from the top show that the numbers of forum switches for $m = 2, 3, 5$ and 10 (left chart), and for $m = 2*, 3*, 5*$ and $10*$ (right chart) are all significantly smaller than that for $m = 30*$, where philosophers use the meeting room in a mutually exclusive style. Likewise, the throughput ratio (shown in the two third charts from the top) also increases as the level of contention increases. All these results show that RA2 performs much better than RA1 does. The two bottom charts show the average number of messages ($\times(N-1)$) needed per request to the meeting room. Although, as analyzed in Theorem 4.10, the worst case message complexity of RA2 is $3(N-1)$, the average message complexity never exceeds $2.3(N-1)$ in the simulation. In contrast, RA1's message complexity is fixed to $2(N-1)$.

In the second experiment we studied the performance of RA2 with respect to the number of fora m. For each m, we considered five levels of contention: 100% (with $\mu_{thinking} = 0$), 50% (with $\mu_{thinking} = 250$ms), 33.3% (with $\mu_{thinking} = 500$ms), 20% (with $\mu_{thinking} = 1000$ms), and 5.9% (with $\mu_{thinking} = 4000$ms). The results are shown in Fig. 8. Compared with the results in Fig. 5, we see that when m increases, RA2's performance decreases more gradually than RA1 does.

In the third experiment we studied how concurrency and delay are affected by various capturing policies, thereby studying the tradeoff between concurrency and delay under various entry policies. Recall that in RA2 only captains are allowed to capture. Capturing increases concurrency at the expense of delaying granting some requests to different fora. By disabling the capturing procedure in RA2, we can see how concurrency and delay are affected by the capturing procedure. Note that in this case RA2's entry policy reduces to RA1's.

We also modified RA2 to allow successors to capture, so as to see if RA2's concurrency can be further increased. It is clear that successors can capture philosophers only in some restricted way, or else livelocks could occur (because two philosophers may repeatedly attend the same forum via the capturing procedure, thereby blocking a third philosopher from attending a different forum). We considered two possible modifications: In the first modification, we allow a successor p_i to capture a philosopher p_j only if p_i does not have a request for a different forum by a third philosopher. We shall use RA2* to denote this modification.

In the second modification, we allow capturing to go at most k levels from a given captain for some constant k. More precisely, we call a philosopher captured by a captain *first-generation successor*. An lth-generation successor attending a forum can capture a philosopher (also by sending it a $Start$ message) requesting the same forum if $l < k$. The captured philosopher is called an $(l+1)$*th-generation successor*, and is also allowed to capture if $(l+1) < k$. We shall use RA2(k) to denote this modification. It can be seen that RA2 can be easily modified to RA2(k). Since RA2(k) allows up to k th-generation successors, RA2(1) reduces to the original version of RA2 that was presented in Sect. 4.1, and RA2(0) reduces to the version of RA2 that does not allow any philosopher to capture.

We measured how a philosopher's request is delayed by measuring the time a philosopher stays in state $waiting$ per request. To see how this delay is affected globally and locally under different entry policies, we measured (1) the *overall delay*, which is the average delay for all philosophers, and (2) the *captain's delay*, which is the average delay for only captains. It is clear that in RA2(k) and RA2*, a round of forum must be initiated by a philosopher entering the forum

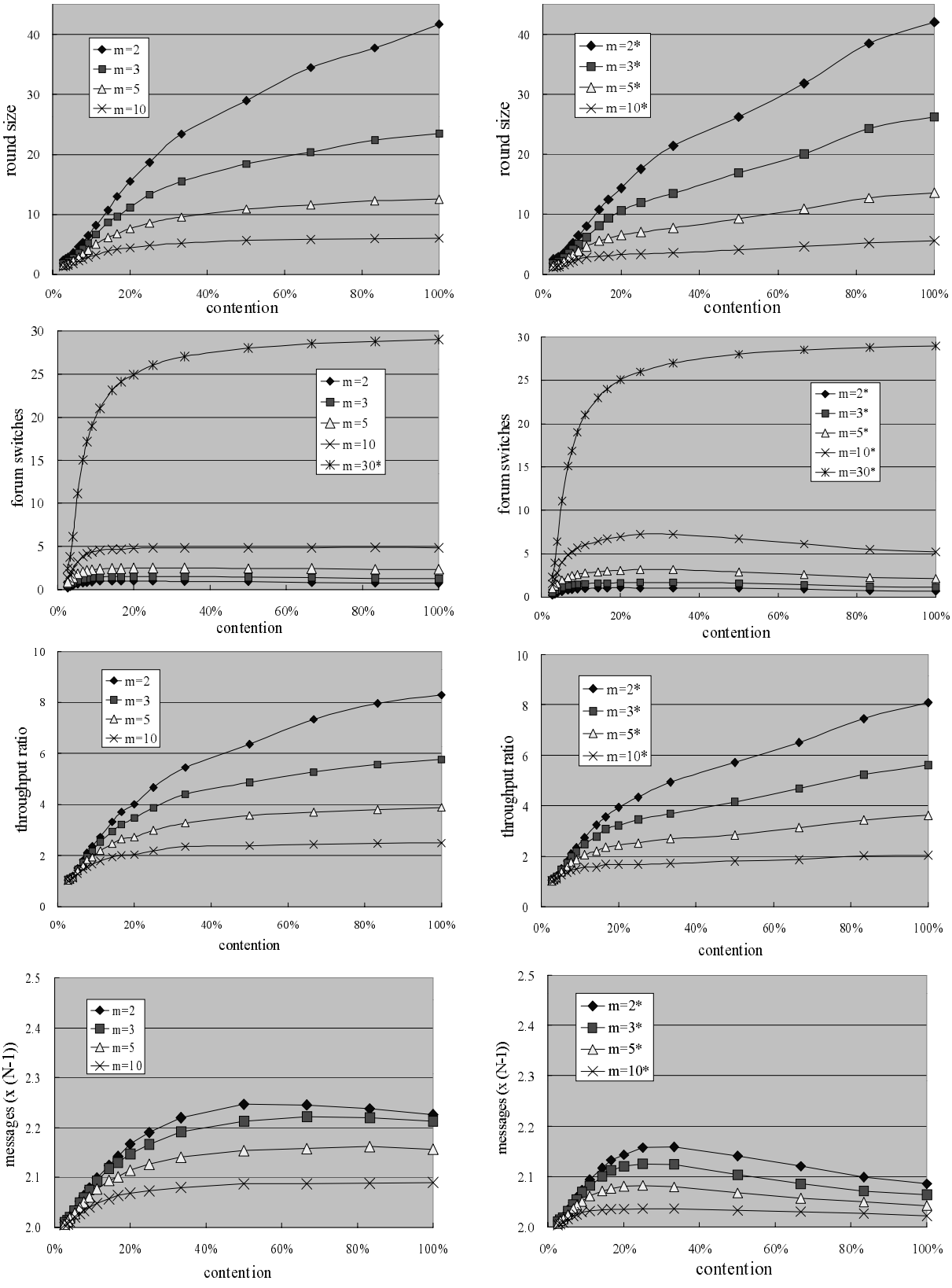

Fig. 7. RA2's performance with respect to contention. The left four charts are for random forum choice, and the right four charts are for fixed forum choice

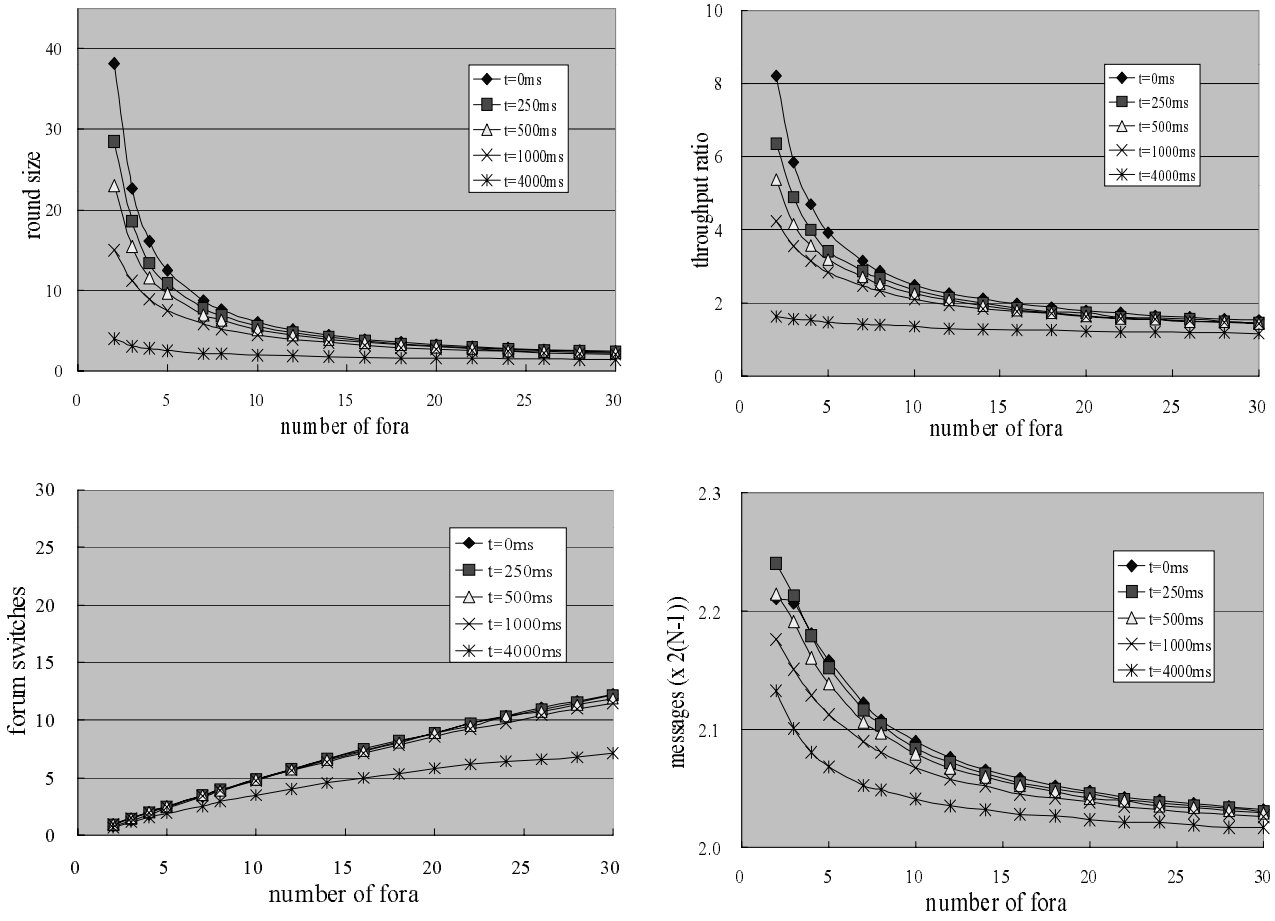

Fig. 8. RA2's performance with respect to the number of fora m. Time t in each chart shows the value of $\mu_{thinking}$

as a captain. So the captain's delay indicates when a round of forum is ongoing and some philosophers are waiting for a different forum, how these philosophers are delayed when we allow late philosophers to jump over them to attend the ongoing forum.

In the first part of this experiment we studied the following algorithms: RA2*, RA2(0), RA2(1), RA2(3), and RA2(10) under different levels of contention. The number of fora m is set to $2*$ (with fixed forum choice). The results are shown in Fig. 9. From the results we see that capturing is crucial in boosting the performance of the system. Without capturing (as in RA2(0)), the measured values for round size, number of forum switches, throughput ratio, overall delay, and captain's delay are all significantly worse than those that allow capturing. Only message complexity is slightly reduced (to $2(N-1)$ as in RA1).

By comparing the results for RA2* and RA2(1), we see that allowing successors to capture while no other philosopher is waiting for a different forum has little help to increase RA2's concurrency. Intuitively, this is because when a philosopher p_i requests a forum X while X is ongoing, the chances that no other philosopher is waiting for a different forum and all philosophers currently attending X are successors are quite low. (If some philosopher attending X is a captain, p_i will be captured by the captain as in RA2.)

Moreover, by comparing the results for RA2(1), RA2(3), and RA2(10), we see that allowing capturing to go more than one level does improve RA2's concurrency. However, the improvement from $k = 1$ to $k > 1$ is not as significant as the improvement from $k = 0$ to $k = 1$. On the other hand, captain's delays for $k = 3$ and $k = 10$ are generally longer than that for $k = 1$ (see the bottom right chart).

In the second part of the concurrency vs. delay experiment, we then studied how k affects RA2(k). For each value of k, we consider four cases: $m = 2*$ with $\mu_{thinking} = 250ms$, $m = 2*$ with $\mu_{thinking} = 1000ms$, $m = 5*$ with $\mu_{thinking} = 250ms$, and $m = 5*$ with $\mu_{thinking} = 1000ms$. The results are shown in Fig. 10. Again, we can see that other than message complexity, RA2(1) performs significantly better than RA2(0) does. For $k \geq 1$, from the top left chart we see that the round size increases as k increases for the following three cases: $m = 2*, \mu_{thinking} = 250ms$, $m = 2*, \mu_{thinking} = 1000ms$, and $m = 5*, \mu_{thinking} = 250ms$. Because the number of philosophers that can potentially attend the same forum is fixed (to N/m), in the above three cases the number of times a philosopher can be captured to re-enter an ongoing forum increases as the capturing level increases. Likewise, the number of forum switches (see the middle left chart) decreases as k increases.

Note that the round size for $m = 5*, \mu_{thinking} = 1000ms$ is roughly unchanged for $k \geq 1$. This is because when the number of philosophers that can potentially attend the same forum is small, and the time philosophers stay in state *thinking* is long, there is little chance for a successor to meet some

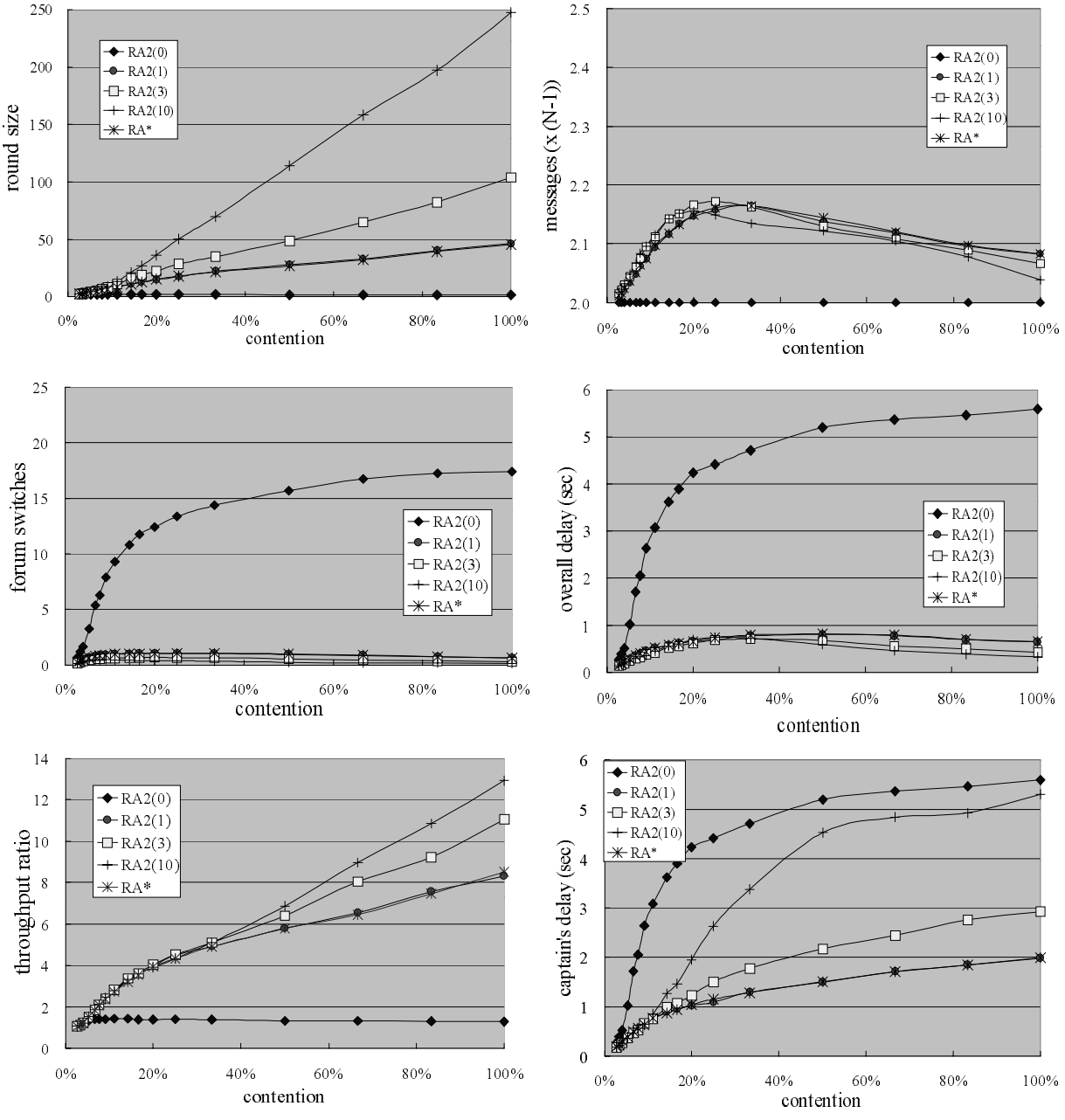

Fig. 9. Performance of RA2* and a family of RA2(k) with respect to contention. The number of fora m is set to $2*$

philosopher requesting the same forum while the successor is in a forum. Similarly, the number of forum switches is roughly not affected by k for $k \geq 1$.

On the other hand, throughput ratio and overall delay stay approximately the same for all $k \geq 1$ in the four cases. This means that the overall performance of the system has almost reached its limit when $k = 1$. Together with the fact that forum size increases as k increases (for the cases $m = 2*$, $\mu_{thinking} = 250ms$; $m = 2*$, $\mu_{thinking} = 1000ms$; and $m = 5*$, $\mu_{thinking} = 250ms$), we can expect that captain's delay will increase as k increases. This is witnessed by the results shown in the bottom right chart.

Finally, message complexity is roughly not affected by k. From the results we can conclude that, in general, $k = 1$ achieves the minimal delay globally and locally while offering the best throughput of the system.

5 Conclusions and future work

We have presented a simple algorithm RA1 for CTP. The algorithm is a straightforward modification from Ricart and Agrawala's algorithm for n-process mutual exclusion. We then showed that although RA1 appears to be fine from various static measures (with $2(N-1)$ in message complexity, forum-switch complexity, and time complexity), it behaves like a mutual exclusion algorithm (where only one philosopher can be in a forum at a time) when the number of fora m the philosophers in CTP may like to hold increases. As analyzed in Sect. 3.2, this performance degradation is due to the combination of the following two facts:

– The entry policy that no philosopher can attend a forum if there is a higher priority request from a different philosopher waiting to attend a different forum.

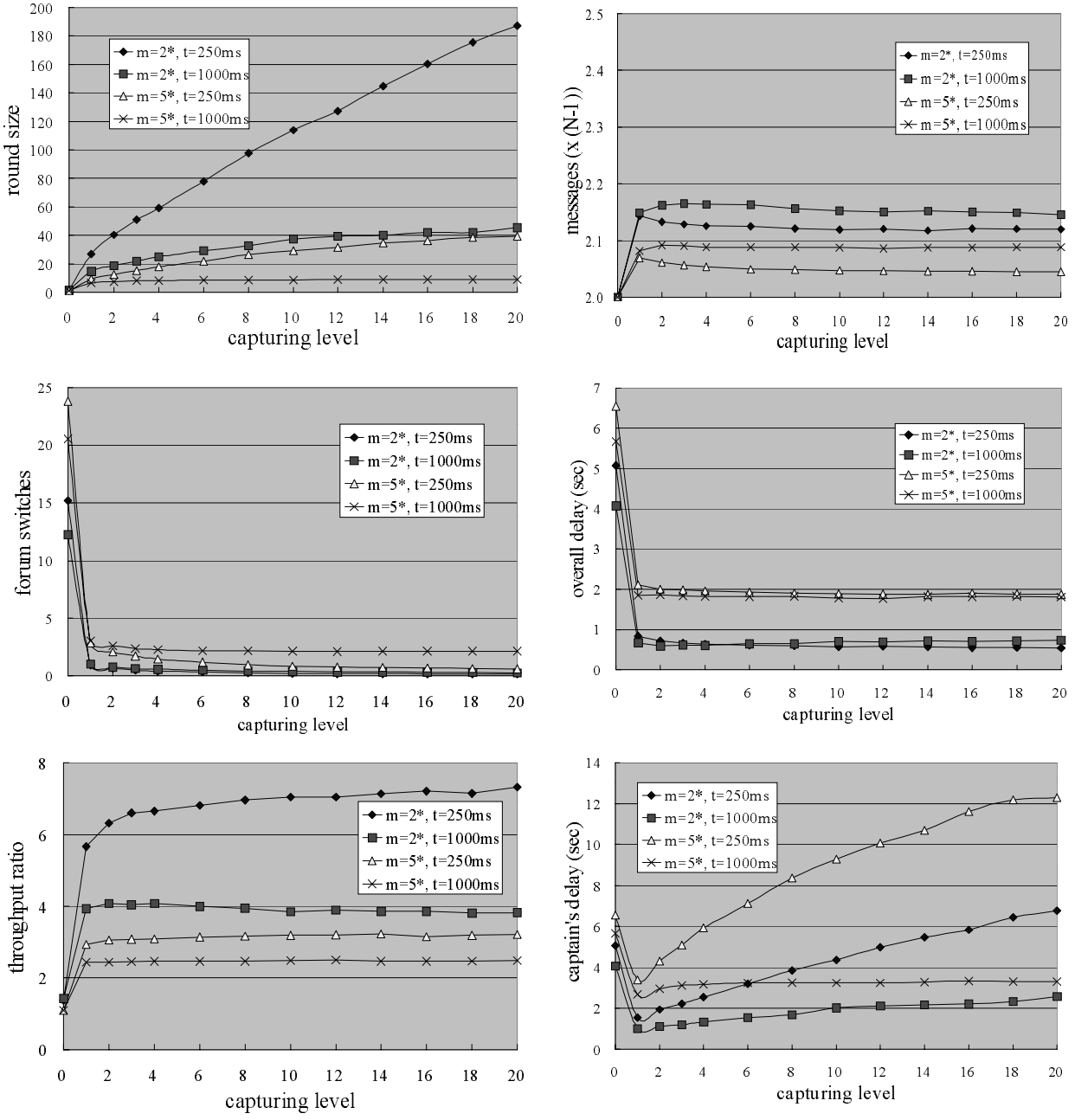

Fig. 10. Performance of RA2(k) with respect to k

– When two groups of philosophers request two different fora simultaneously, the "order" of their requests enforced by the system (that is used to determine the priorities of the requests) is likely to place philosophers of different groups, rather than philosophers of the same group, in an adjacent order.

Based on this finding we then proposed a weaker entry policy as follows, and presented a new algorithm RA2 to implement this policy:

– No philosopher p_j can attend a forum X if there is a higher priority request from some p_k waiting to enter a different forum, unless there is another p_i already in X such that p_i's request has priority higher than p_k's request.

RA2 requires N to $3(N-1)$ messages per entry to the meeting room, and has forum-switch complexity $2(N-1)$, and time complexity $O(N^2)$. Although these static measures do not differ much from those of RA1, RA2 performs significantly better than RA1 does at run time.

Based on RA2, we have also studied several other entry policies. Specifically, we allow capturing in RA2 to go to k generations so that lth-generation successors are allowed to capture if $l < k$ for some constant k. Our experimental results indicate that, in general, $k = 1$ (i.e., RA2's original entry policy) achieves the minimal delay globally and locally while offering the best throughput of the system.

The comparison between RA1 and RA2 also highlights several directions for future work. For example, in the worst case, RA2 needs $N-1$ more messages per entry to the meet-

ing room than RA1 does (although in the simulation RA2's average message complexity is no more than $2.3(N-1)$). Moreover, each request in RA2 bears a vector timestamp, while only a timestamp is needed in RA1. The use of vector timestamps compromises the scalability of RA2. Thus, it will be interesting to see if RA2 can be further improved to reduce its message complexity and message size. It will also be interesting to see, from the information structure point of view, minimum information required for exchange between processes for various entry policies.

In the paper we have chosen Ricart and Agrawala's algorithm as our first step for solving group mutual exclusion in computer networks. As we said before, the choice came from the observation that the algorithm can be straightforwardly extended to group mutual exclusion, while no other mutual exclusion algorithms we are aware of exhibit this simplicity in the extension. Nevertheless, this simple algorithm suffers a severe performance degradation and so it is interesting to see why this degradation occurs and how it can be avoided. As discussed above, the degradation is due to the fact that when two groups of processes request for critical section simultaneously, the "order" of their requests enforced by the system is likely to place processes of different groups, rather than processes of the same group, in an adjacent order. So to increase concurrency, requests cannot be granted entirely based on this order. We believe that this finding can be applied to other "permission-based" approaches to group mutual exclusion.

In the literature numerous algorithms have been proposed for mutual exclusion (see, for example, brief surveys from [23,21,24]) and for k-exclusion (see, e.g., [19,25,4,3, 13,8]) in computer networks. These algorithms can be broadly classified into two categories: *token based* and *permission based* [21]. Token based algorithms achieve mutual exclusion through a unique token competed for by the processes, while in permission based algorithms a process seeks permissions to enter the critical section from some set of processes called a *quorum* [9]. In general, token based algorithms produce less message traffic, but have long minimum synchronization delay. (*Minimum synchronization delay* is the minimum delay, measured by message transmission time, from the time a process invokes a mutual exclusion request to the time it enters the critical section.) In contrast, permission based algorithms have short minimum synchronization delay, and are more fault-tolerant because a quorum usually consists of a subset of the processes in the system. Its message complexity depends on the size of the quorum a process uses. For example, the *finite projective planes* quorum system proposed by Maekawa [17] has quorum size $\sqrt{N}$, and so a mutual exclusion algorithm based on this quorum system requires only $O(\sqrt{N})$ messages per entry to the critical section. (For comparison, Ricart and Agrawala's algorithm needs $O(N)$ messages, and cannot tolerate any site failure.)

In light of the above discussion, it might be interesting to investigate quorum-based algorithms for group mutual exclusion. As discussed above, a process must acquire a quorum (i.e., obtains permission from every member of the quorum) in order to enter the critical section. A quorum member gives permission to only one process at a time. So a quorum system for mutual exclusion requires that every two quora in the system intersect, and a quorum system for k-exclusion requires that any collection of $k+1$ quora contains a pair of intersecting quora.

It is clear that that quorum systems for mutual exclusion are not suitable for group mutual exclusion, for otherwise, only one process can be in the critical section at a time. Similarly, quorum systems for k-exclusion cannot be applied to group mutual exclusion, for otherwise, two processes from different groups might successfully acquire two disjoint quora simultaneously and then both enter the critical section. Thus, quorum systems for group mutual exclusion must be completely redesigned. Our future work will focus on the design of quorum systems for group mutual exclusion.

Acknowledgements. The author would like to thank Kuen-Pin Wu, Wen-Jian Tsai, and Jen-Yi Liao for comments on the earlier work of this research. The author would also like to thank Vassos Hadzilacos and the anonymous referees; their comments and suggestions have substantially improved the content and the presentation of the paper. A preliminary version of this paper appeared in *Proc. 13th International Symposium on DIStributed Computing (DISC99), Lecture Notes in Computer Science 1693*, pp. 195–209, 1999.

Appendix A: Definition on concurrent occupancy

The definition of "concurrent occupancy" given in the paper is due to Keane and Moir [14], but the term was recently coined by Hadzilacos [10]. In the original specification of CTP [12], I used the term "concurrent entering" to exclude solutions to CTP that may impose unnecessary synchronization among philosophers attending a forum when no one else is interested in a different forum. The definition given in [12] is as follows:

> **Concurrent Entering**: If some philosophers are interested in a forum and no philosopher is interested in a different forum, then the philosophers can attend the forum concurrently.

Unnecessary synchronization occurs, for example, in solutions that directly or indirectly apply ordinary mutual exclusion algorithms to solve the problem. The use of such algorithms forces philosophers to wait for one another in order to *enter* the meeting room, even if no other philosopher is interested in a different forum. However, the phrase "attend the forum concurrently" although suggestive is somewhat vague. Keane and Moir [14] later gave a different interpretation of the definition as seen in Sect. 1:

> **Concurrent Occupancy**: If some philosopher p has requested a forum X and no philosopher is currently attending or requesting a different forum, then p can attend X without waiting for other processes to leave the forum.

The definition they gave is weaker than what I intended. In particular, their definition precludes a direct application of ordinary mutual exclusion algorithms to CTP, but does not preclude the use of such algorithms as a primitive.

The distinction between these two definitions was recently clarified by Hadzilacos [10]. He called Keane and Moir's definition "concurrent occupancy", and gave a precise definition for "concurrent entering" in a shared-memory model as follows:

If a philosopher p requests a forum and no philosopher requests a different forum, then p enters the meeting room within a bounded number of its own steps.

A fundamental difference between shared memory and message passing is that in shared memory, a process p can communicate with another process q (by reading their shared variables) without q's step, while this is not possible in message passing. Since communication is necessary in group mutual exclusion, no message-passing algorithm for CTP can satisfy Hadzilacos's definition of concurrent entering. So the paper adopted Keane and Moir's definition of "concurrent occupancy". As commented above, their definition precludes a direct application of ordinary mutual exclusion algorithms to CTP, but does not preclude the use of such algorithms as a primitive. Nevertheless, the algorithms presented in the paper do not use such primitives.

References

1. Y. Afek, D. Dolev, E. Gafni, M. Merritt, N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. ACM Transactions on Programming Languages and Systems, 16(3):939–953, 1994
2. K. Alagarsamy, K. Vidyasankar. Elegant solutions for group mutual exclusion problem. Technical report, Dept. of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, 1999
3. R. Baldoni, B. Ciciani. Distributed algorithms for multiple entries to a critical section with priority. Information Processing Letters, 50(3):165–172, 1994
4. S. Bulgannawar, N.H. Vaidya. A distributed K-mutual exclusion algorithm. In Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95), pp. 153–160, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press
5. P.J. Courtois, F. Heymans, D.L. Parnas. Concurrent control with readers and writers. Communications of the ACM, 14(10):667–668, 1971
6. F.N. David, D.E. Barton. Combinatorial chance. London : Charles Griffin & Co., 1962
7. M.J. Fischer, N.A. Lynch, J.E. Burns, A. Borodin. Resource allocation with immunity to limited process failure (preliminary report). In 20th Annual Symposium on Foundations of Computer Science, pp. 234–254, San Juan, Puerto Rico, 29–31 October 1979. IEEE
8. S. Fujita. A quorum based k-mutual exclusion by weighted k-quorum systems. Information Processing Letters, 67(4):191–197, 1998
9. H. Garcia-Molina, D. Barbara. How to assign votes in a distributed system. Journal of the ACM, 32(4):841–860, 1985
10. V. Hadzilacos. A note on group mutual exclusion. In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island, August 2001. ACM Press
11. C.A.R. Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666–677, 1978
12. Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 51–60, Puerto Vallarta, Mexico, June 1998. ACM Press. Full paper in Distributed Computing, 13(4):189-206, 2000
13. H. Kakugawa, S. Fujita, M. Yamashita, T. Ae. A distributed k-mutual exclusion algorithm using k-coterie. Information Processing Letters, 49(4):213–218, 1994
14. P. Keane, M. Moir. A simple local-spin group mutual exclusion algorithm. In Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 23–32. ACM Press, 1999
15. L. Lamport. Time, clocks and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978
16. N.A. Lynch. Distributed Algorithms. Morgan-Kaufmann, 1996
17. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. ACM Transactions on Computer Systems, 3(2):145–159, 1985
18. F. Mattern. Virtual time and global states in distributed systems. In Proceedings of the International Workshop on Parallel and Distributed Algorithms, pp. 215–226, Gers, France, 1989. Amsterdam: North-Holland 1989
19. K. Raymond. A distributed algorithm for multiple entries to a critical section. Information Processing Letters, 30:189–193, 1989
20. M. Raynal. Algorithms for Mutual Exclusion. MIT Press, Cambridge, MA, 1986
21. M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. ACM Operating Systems Review, SIGOPS, 25(2):47–50, 1991
22. G. Ricart, A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM, 24(1):9–17, 1981
23. B.A. Sanders. The information structure of distributed mutual exclusion algorithms. ACM Transactions on Computer Systems, 5(3):284–299, 1987
24. M. Singhal. A taxonomy of distributed mutual exclusion. Journal of Parallel and Distributed Computing, 18(1):94–101, 1993
25. P.K. Srimani, R.L.N. Reddy. Another distributed algorithm for multiple entries to a critical section. Information Processing Letters, 41(1):51–57, 1992

Yuh-Jzer Joung received his B.S. in electrical engineering from the National Taiwan University in 1984, and his M.S. and Ph.D. in computer science from the State University of New York at Stony Brook in 1988 and 1992, respectively. He was a visiting scientist at Laboratory for Computer Science, Massachusetts Institute of Technology in 1999-2000. He is currently a professor and chair in the Department of Information Management at the National Taiwan University, where he has been a faculty member since 1992. His main research interests are in the area of distributed computing, with specific interests in process scheduling, multiparty interaction, fairness, and mutual exclusion.