Asynchronous group mutual exclusion in ring networks

K.-P.Wu and Y.-J.Joung

Abstract: In group mutual exclusion solutions for shared-memory models and complete message-passing networks have been proposed. These solutions, however, cannot be straightforwardly and efficiently converted to ring networks where each process can only communicate directly with its two neighbouring processes. As rings are also a popular network topology, the paper is focused on ring networks. An efficient and highly concurrent distributed algorithm for the problem is presented.

1 Introduction

The design issues for imposing mutual exclusion on different groups of processes in accessing a resource while exploring concurrency among processes of the same group in sharing the resource have recently been modelled by Joung [1] as the congenial talking philosophers (CTP): a set of N philosophers $p_0, p_1, \dots, p_{N-1}$ spend their time thinking alone and talking in a forum. Initially, all philosophers are thinking. From time to time, when a philosopher is tired of thinking, it wishes to attend a forum of its choice. Given that there is only one meeting room, 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 same forum is already in the meeting room (and in this case the philosopher joins this ongoing forum). We assume that when a philosopher has attended a forum, it spends an unpredictable but finite amount of time in the forum. After a philosopher leaves a forum (i.e. exits the meeting room), it returns to thinking. The problem is to design an algorithm for the philosophers satisfying the following requirements:

mutual exclusion: if a philosopher is in a forum, then no other philosopher can be in a different forum simultaneously;

bounded delay: a philosopher attempting to attend a forum will eventually succeed;

concurrent entering: if some philosophers are interested in a forum and no philosopher is interested in a different forum, then the philosophers interested in the same forum can concurrently enter the meeting room to hold the forum.

As discussed in [2], ‘concurrent entering’ is used to prevent an unnecessary synchronisation on philosophers in attending a forum when no other philosopher is interested in a different forum. For example, in a token-ring system we can easily construct a solution as follows: when a philosopher p_i interested in a forum X obtains the token, it initiates X and sends out an invite message along the ring to inform other philosophers interested in X to attend X . To close the forum, a clean-up message is issued by p_i to make sure no philosopher is still in X before p_i releases the token. The use of a single token easily guarantees mutual exclusion and bounded delay, but it also makes concurrent entering impossible. This is because if two philosophers wish to attend X simultaneously, then one of them must enter the meeting room *sequentially* after the other.

Solutions to CTP for shared-memory models and complete message-passing networks have been proposed [1, 2]. They, however, cannot be straightforwardly and efficiently converted to ring networks where each philosopher can only communicate directly with its two neighbouring philosophers. As rings are also a popular network topology, in this paper we focus the problem on ring networks (Fig. 1) where the N philosophers $p_0, p_1, \dots, p_{N-1}$ are connected via a ring network so that each philosopher p_i can only send messages directly to its immediate successor p_{i+1} . (Unless stated, otherwise, additions and subtractions on indices of philosophers are to be interpreted modulo N .) We shall assume that philosophers are distinguished by their unique IDs, and that every

Fig. 1 Ring network

© IEE, 2000

IEE Proceedings online no. 20000162

DOI: 10.1049/ip-cdt:20000162

Paper first received 16th February and in revised form 17th September 1999

K.-P. Wu is with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
E-mail: kpw@orchid.ee.ntu.edu.tw

Y.-J. Joung is with the Department of Information Management, National Taiwan University, Taipei, Taiwan
E-mail: joung@ccms.ntu.edu.tw

message will eventually be delivered to its destination. However, we do not require FIFO delivery of messages. As we shall see in Section 4, assuming FIFO delivery results in a fewer ‘context switches’ complexity in our algorithm. With a slight modification, however, the FIFO assumption can be lifted without affecting the complexity, and so makes the algorithms suitable for more general models.

2 Straightforward solution

A straightforward solution to CTP, which we shall refer to as CTP-Ring1, can be easily devised as follows. Each philosopher p_i is in one of the following three states: *thinking*, meaning that it is not interested in any forum; *waiting*, meaning that it is waiting for a forum; and *talking*, meaning that it is in a forum (Fig. 2). p_i maintains a variable SN recording the maximum sequence number seen by the philosopher. The variable will be maintained similar to Lamport’s logical clock [3]. That is, SN is initialised to 0 and, whenever p_i learns of a sequence number larger than its own SN , it advances its SN to that number. Furthermore, p_i increments SN by 1 when it wishes to attend a forum.

Initially, p_i is in state *thinking*. When it wishes to attend a forum X , it enters state *waiting*, increments its SN by 1, and sends a request message $Req(\langle i, sn \rangle, X)$ to its successor p_{i+1} , where sn is the new value of p_i ’s SN . Philosopher p_i remains in state *waiting* until its request $Req(\langle i, sn \rangle, X)$ is returned. Then, p_i enters the meeting room to attend X . After finishing the forum, p_i returns to state *thinking*.

When p_{i+1} receives a request message $Req(\langle j, sn_j \rangle, X)$ from p_i , it either forwards the request to its successor p_{i+2} , or detains the request in its message queue; the decision depends on p_{i+1} ’s state. p_{i+1} forwards the request if one of the following conditions is satisfied: (1) it is in state *thinking*; (2) it is also interested in X ; or (3) it is interested in a different forum and has a priority lower than p_j . A philosopher’s priority is assigned as follows: the priority is set to $\langle j, sn_j \rangle$ when the philosopher issues a request $Req(\langle j, sn_j \rangle, X)$, and is reset to a minimal value $\langle j, \infty \rangle$ when it finishes X . A priority $\langle j, sn_j \rangle$ is higher than $\langle k, sn_k \rangle$, denoted by $\langle j, sn_j \rangle > \langle k, sn_k \rangle$ if, and only if, $sn_j < sn_k$, or $sn_j = sn_k$ and $j < k$. (Thus, a philosopher’s priority in state *thinking* is always lower than that of a philosopher interested in a forum.)

On the other hand, p_{i+1} detains the request if none of the above three conditions is satisfied; that is, p_{i+1} is interested in a different forum Y and has a priority higher than $\langle j, sn_j \rangle$. p_{i+1} detains the request until it has finished Y . Then, p_{i+1} forwards the request (and all other requests it has held) to p_{i+2} .

It is easy to see that CTP-Ring1 requires N messages per entry to the meeting room. The algorithm also satisfies the three requirements of CTP. For mutual exclusion, observe

Fig. 2 State transition diagram of a philosopher in CTP-Ring1

that by the way sequence numbers are maintained, if a philosopher p_j requests a forum after it has received $Req(\langle i, sn \rangle, X)$ issued by p_i , then p_j ’s priority must be lower than p_i ’s. So, when p_j ’s request circulates to p_i , if the request is not for X then it will be detained by p_i until p_i has finished X . Therefore, p_j cannot attend a different forum while p_i is in X . Moreover, given that priorities are unique, when two philosophers request different fora concurrently, the request issued by the low-priority philosopher will be detained by the high-priority one until the latter has finished a forum. So, the two philosophers cannot be in different fora simultaneously. Because neither logically dependent requests nor concurrent requests can violate mutual exclusion, mutual exclusion is guaranteed.

For bounded delay, observe that a philosopher releases all detained messages to its successor when it finishes a forum. Given that a philosopher spends only a finite time in a forum and that every message is eventually delivered, a philosopher issuing a request will eventually receive the request and, so, will eventually attend a forum.

For concurrent entering, observe that philosophers will not detain one another’s request if they are interested in the same forum. Given that philosophers not interested in a forum will not detain any request, if some philosophers are interested in a forum and no philosopher is interested in a different forum, then the philosophers can enter the meeting room concurrently.

2.1 Simulation results

In the simulation, we have set up a system of N philosophers and m fora. Each time a philosopher wishes to attend a forum, it randomly chooses one of the m fora to attend, and the choice follows a uniform distribution. The time a philosopher stays in states *thinking* and *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} . We measure the following values:

- The average time a philosopher spends in waiting for a forum: that is, the average waiting time.
- The average number of ‘context switches’ a philosopher has to wait per request, where a *context switch* (or *forum switch*) occurs when the next entry to the meeting room is for a forum different from the current one. The greater the number of context switches the longer the waiting time.
- The average size of a round of X ; where a *round of X* is a maximal set of consecutive entries to the meeting room that are all to attend X .
- The average capacity; that is, on an average, the maximum number of philosophers that can be in the meeting room simultaneously per round.
- The average number of messages required per entry to the meeting room.
- Throughput; that is, the average number of entries to the meeting room per second.

The simulation program is written in Java using JDK V1.02 [4]. Tables 1–8 summarise some of our simulation results. For comparison, we have also measured the behaviour of the algorithm when philosophers use the meeting room in a mutually exclusive style. This is done by designating one unique forum to each philosopher. In the tables we use $m = N^*$ to denote this scenario. We have also set up the case $m = 1$ where maximum concurrency should be allowed as no two requests will ever conflict. The average message transmission time μ_{link_delay} is set to two different values, one is negligible compared to $\mu_{thinking}$ and

$\mu_{talking}$, while the other is not. It is not difficult to see that the settings represent a very high contention situation to the meeting room. From Tables 1 and 5 we can see that CTP-Ring1 provides virtually no concurrency. For example, when $m=3$, it is likely that one third of the pending requests are targeted at the same forum. However, the simulation results indicate that the behaviour of the system is only slightly better than the case $m=30^*$ where the philosophers use the meeting room in a mutually exclusive style.

Fig. 3 New state transition diagram of a philosopher

Table 1: Simulation results for algorithm CTP-Ring1

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	20.12	6108.46	6816.79	7292.95	7605.84	7873.7
Context switches	0	19.08	23.00	25.91	27.41	28.86
Round size	NA	1.51	1.25	1.11	1.05	1.00
Capacity	30	1.49	1.24	1.11	1.05	1.00
Messages / entry	30	30	30	30	30	30
Throughput (entry/s)	92.13	4.66	4.19	3.95	3.79	3.67

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=2$ ms

Table 2: Simulation results for algorithm CTP-Ring2

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	82.62	1635.33	2195.67	3225.78	4397.95	7973.73
Context switches	0	2.01	3.03	5.42	9.53	28.87
Round size	NA	14.27	9.47	5.29	3.01	1.00
Capacity	30	12.62	8.62	4.97	2.90	1.00
Message / entry	60	60	60	60	60	60
Throughput (entry/s)	76.90	15.39	11.95	8.48	6.36	3.62

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=2$ ms

Table 3: Simulation results for algorithm CTP-Ring3

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	84.92	1538.9	2124.01	3227.53	4398.81	7899.85
Context switches	0	1.93	2.91	5.34	9.44	28.87
Round size	NA	14.64	9.75	5.37	3.05	1.00
Capacity	30	13.11	8.95	5.06	2.92	1.00
Messages / entry	60	60	60	60	60	60
Throughput (entry/s)	76.66	16.06	12.22	8.46	6.38	3.65

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=2$ ms

Table 4: Simulation results for algorithm CTP-Ring4

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	85.36	1535.52	2084.81	3151.52	4404.95	7822.91
Context switches	0	1.95	2.93	5.32	9.51	28.87
Round size	NA	14.71	9.80	5.40	3.02	1.00
Capacity	30	13.27	9.04	5.08	2.91	1.00
Messages / entry	60	46.53	43.41	42.55	44.45	60
Throughput (entry/s)	76.30	16.23	12.51	8.66	6.36	3.69

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=2$ ms

Table 5: Simulation results for algorithm CTP-Ring1

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	778.53	8699.99	10575.3	12305.5	13434	15827.8
Context switches	0	19.18	23.18	26.12	27.41	29.00
Round size	NA	1.50	1.25	1.11	1.06	1.00
Capacity	19	1.39	1.20	1.09	1.05	1.00
Messages / entry	30	30	30	30	30	30
Throughput (entry/s)	27.64	3.33	2.76	2.38	2.18	1.86

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=20$ ms

Table 6: Simulation results for algorithm CTP-Ring2

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	1659.17	3766.88	4694.26	6172.01	8184.44	15815.3
Context switches	0	1.82	2.86	5.27	9.36	28.99
Round size	NA	15.22	9.75	5.35	3.04	1.00
Capacity	17	6.59	5.09	3.39	2.28	1.00
Messages / entry	60	60	60	60	60	60
Throughput (entry/s)	15.26	7.35	5.99	4.63	3.53	1.86

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=20$ ms

Table 7: Simulation results for algorithm CTP-Ring3

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	1657.47	3345.44	4252.13	5736.37	7481.06	15400
Context switches	0	1.71	2.75	5.08	9.22	28.99
Round size	NA	16.43	10.21	5.56	3.09	1.00
Capacity	17	7.39	5.51	3.62	2.36	1.00
Messages / entry	60	60	60	60	60	60
Throughput (entry/s)	15.26	8.21	6.57	4.96	3.85	1.91

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=20$ ms

Table 8: Simulation results for algorithm CTP-Ring4

Average	$m=1$	$m=3$	$m=5$	$m=10$	$m=20$	$m=30^*$
Waiting time (ms)	1657.47	2781.95	3459.63	4655.31	6257.89	15675.9
Context switches	0	1.73	2.79	5.17	9.29	28.99
Round size	NA	16.32	10.14	5.48	3.07	1.00
Capacity	17	7.71	5.86	3.74	2.39	1.00
Messages / entry	60	45.65	43.26	42.41	44.16	60
Throughput (entry/s)	15.26	9.71	7.96	6.04	4.57	1.88

$N=30$, $\mu_{thinking}=50$ ms, $\mu_{talking}=250$ ms, $\mu_{link_delay}=20$ ms

3 Improved algorithm

The low concurrency of CTP-Ring1 is due to the fact that philosophers' requests are granted according to their priorities: a low-priority philosopher must enter the meeting room after a high-priority one. 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 between them, then p_i and p_j cannot attend the same forum concurrently because the low-priority philosopher p_j must wait for p_k to finish a forum before it can

attend a forum. To resolve this, we let p_i issue another message along the ring to 'capture' p_j (and other philosophers that are also interested in the same forum) so that the captured philosophers can attend the forum concurrently with the captor.

To do so, we introduce a new state *checking* for the philosophers (Fig. 3). As before, a philosopher's priority is set to $\langle i, sn \rangle$ when it issues a request $Req(\langle i, sn \rangle, X)$ (and $\langle i, sn \rangle$ is also said to be the priority of the request). However, the priority may be 'upgraded' (but at most once) before the philosopher attends X . The priority is reset to a minimal

value $\langle i, \infty \rangle$ when the philosopher returns to state *thinking*.

Like CTP-Ring1, when a philosopher p_i wishes to attend a forum X , it enters state *waiting*, increments its SN by 1, and sends a request $Req(\langle i, sn \rangle, X)$ to its successor p_{i+1} , where sn is the new value of p_i 's SN . Another philosopher p_j in the ring, upon receiving the request, forwards the request if it is also interested in X or has a priority lower than $\langle i, sn \rangle$; otherwise, p_j detains the request in its message queue until it has finished a forum. When the request is returned to p_i , p_i enters state *checking* to issue a confirmation message of the form $Conf/Capt(\langle i, sn \rangle, X)$ to its successor p_{i+1} . The purpose of the message is twofold: to capture philosophers that are waiting for X , and to make sure that no philosopher is still in a different forum, p_i attends X when the message is returned to p_i .

When p_j receives $Conf/Capt(\langle i, sn \rangle, X)$, where $j \neq i$, it processes the message similarly to request messages: p_j forwards the message if it is also interested in X or has a priority lower than $\langle i, sn \rangle$; otherwise, p_j detains the message. Note that all messages (requests and confirmations) detained by a philosopher are released when the philosopher exits a forum. In addition, if p_j is also interested in X , is in state *waiting*, and has a priority lower than $\langle i, sn \rangle$, then p_j is *captured* by p_i . In this case, p_j assumes p_i 's priority and also enters state *checking* to issue a confirmation message of the form $Conf(j, \langle i, sn \rangle, X)$ along the ring. (p_j 's request message circulating somewhere in the ring then becomes obsolete.) Unlike p_i 's confirmation message, p_j 's confirmation message serves only one purpose: to make sure that no philosopher is still in a different forum. Therefore, a philosopher receiving p_j 's confirmation message processes the message similarly to the other form of confirmation messages, except that the new form of confirmation messages cannot capture the receiving philosophers. This prevents p_j from capturing p_i after p_i has attended X and has made a new request to re-attend X , thereby preventing p_i and p_j from capturing each other repeatedly while blocking another forum indefinitely.

p_i enters state *talking* to attend X when its confirmation message is returned. After p_i has finished the forum, it forwards all the messages it has detained to p_{i+1} and returns to state *thinking*. Note that because some low-priority philosophers may have been captured by p_i to concurrently attend X , after p_i has left X , some philosophers may still be in X while another p_k with a priority higher than the captured philosophers is waiting for a different forum Y . Because p_i will release all messages it has detained when it leaves X , p_k may have received its own request after p_i has left X but before all its captured philosophers have left X . Therefore, p_k cannot immediately enter state *talking* to attend Y when it receives its own request. Instead, it must initiate a confirmation message to make sure that all p_i 's captured philosophers have left X . Similarly, any philosopher captured by p_k must also initiate a confirmation message along the ring before attending a forum.

We refer to the new algorithm as CTP-Ring2, and its complete code is given in Fig. 4. The algorithm is represented by a CSP-like repetitive command of the form [4]:

$$*[g_1 \rightarrow s_1 \square g_2 \rightarrow s_2 \square \dots \square g_k \rightarrow s_k]$$

The variables local to each p_i are given as follows:

- *state*: the state of p_i . It is initialised to *thinking*.
- *SN*: the sequence number maintained by p_i . It is initialised to 0.
- *target*: the forum p_i wishes to attend, or $\perp$ otherwise. It is initialised to $\perp$.

- *priority*: p_i 's priority. It is initialised to a minimal value $\langle i, \infty \rangle$.
- *message_queue*: the queue of messages detained by p_i . It is initialised to empty.

4 Correctness of CTP-Ring2 and its fine tuning

We now prove the correctness of CTP-Ring2. Recall from the algorithm that for a philosopher p_i to attend X , p_i has to issue a request for X and then a confirmation message. We say that the request is *granted* when the confirmation message has returned to p_i . p_i is *waiting* for X after it has issued a request for X but before the request has been granted. p_i is *interested* in X if it has issued a request for X but has not yet completed X .

Moreover, recall that a philosopher p_i attends forum X only if:

1. it is in state *checking* and receives its own message $Conf/Capt(\langle i, sn \rangle, X)$, or
2. it is in state *checking* and receives its own message $Conf(i, \langle j, sn \rangle, X)$.

In the first case, we say that p_i enters *checking* as a *captor*, whereas in the later case p_i enters *checking* as a *captive*.

Theorem 4.1: The algorithm guarantees mutual exclusion.

Proof: Let p_i be the philosopher with the highest priority, and assume that p_i has issued $Req(\langle i, sn_i \rangle, X)$. Observe that no other p_j interested in a different forum can be in the meeting room because p_j 's request cannot pass through p_i until p_i has left X . Moreover, p_i 's request will return to p_i because no philosopher can detain the request. Similarly, when p_i subsequently issues a confirmation $Conf/Capt$, the message can also circulate along the ring and return to p_i , thereby allowing p_i to attend X . Every p_k that is captured by p_i while the message is circulating then assumes p_i 's priority. Since any p_j that is interested in a different forum has its request detained by p_i , p_j must have not yet issued a confirmation message when p_i captures p_k . So when p_j 's confirmation message arrives at p_k , p_k has already assumed p_i 's priority and so will detain the message until p_k has left X . Therefore, while p_i or any of its captives is in X , no other p_j can attend a different forum.

Theorem 4.2: The algorithm guarantees bounded delay.

Proof: This follows from the fact for every request $Req(\langle i, sn_i \rangle, X)$, there are at most a finite number of requests that can have a priority higher than $\langle i, sn_i \rangle$, and that the request with the highest priority is eventually granted.

Theorem 4.3: The algorithm permits concurrent entering.

Proof: This follows from the fact that if some philosophers are interested in a forum X and no philosopher is interested in a different forum, then no philosopher can detain the requests for X , and so the philosophers interested in X can concurrently enter the meeting room to hold X .

It is clear that CTP-Ring2 requires $2N$ messages per entry to the meeting room. Moreover, if messages need not be delivered in the order sent, then a philosopher may need to wait for $O(N^2)$ 'context switches' before entering the meeting room (CTP-Ring1 also has the same complexity). To see this, suppose that p_i has issued $Req(\langle i, sn \rangle, X)$. Then, while the request is in transit to, say, p_{i+1} , another p_j could also have issued a request $Req(\langle j, sn' \rangle, Y)$ with $sn' < sn$ (and so p_j 's priority is higher than p_i 's). Since messages do not need to be delivered in the order sent, p_j 's request could have been returned to p_j while p_i 's message is still in transit


```

1  *[wish to attend a forum of X →
2      SN := SN + 1;
3      target := X;
4      priority := ⟨i, SN⟩;
5      state := waiting;
6      send Req(⟨i, SN⟩, X) to pi+1;
7  □ receive Req(⟨j, sn⟩, Y) →
8      SN := max(SN, sn);
9      [i = j ∧ state = waiting ∧ priority = ⟨j, sn⟩ →
10         state := checking;
11         send Conf/Capt(⟨i, sn⟩, target) to pi+1;
12     □ i ≠ j ∧ (target = Y ∨ priority < ⟨j, sn⟩) →
13         send Req(⟨j, sn⟩, Y) to pi+1;
14     □ i ≠ j ∧ target ≠ Y ∧ priority > ⟨j, sn⟩ →
15         add Req(⟨j, sn⟩, Y) to message_queue;
16     □ i = j ∧ (state ≠ waiting ∨ priority ≠ ⟨j, sn⟩) →
17         skip; ] /* pi receives its own obsolete request */
18 □ receive Conf/Capt(⟨j, sn⟩, Y) →
19     [i = j → /* receive pi's own confirmation message */
20         state := talking;
21         attend a forum of target;
22     □ i ≠ j ∧ target ≠ Y ∧ priority > ⟨j, sn⟩ →
23         add Conf/Capt(⟨j, sn⟩, Y) to message_queue;
24     □ i ≠ j ∧ (target = Y ∨ priority < ⟨j, sn⟩) →
25         send Conf/Capt(⟨j, sn⟩, Y) to pi+1;
26         [target = Y ∧ state = waiting ∧ priority < ⟨j, sn⟩ →
27             state := checking; /* captured by pj */
28             priority := ⟨j, sn⟩; /* assume the captor's priority */
29             send Conf(i, ⟨j, sn⟩, target) to pi+1; ]
30 □ receive Conf(j, ⟨k, sn⟩, Y) →
31     [i = j → /* receive pi's own confirmation message */
32         state := talking;
33         attend a forum of target;
34     □ i ≠ j ∧ target ≠ Y ∧ priority > ⟨k, sn⟩ →
35         add Conf(j, ⟨k, sn⟩, Y) to message_queue;
36     □ i ≠ j ∧ (target = Y ∨ priority < ⟨k, sn⟩) →
37         send Conf(j, ⟨k, sn⟩, Y) to pi+1; ]
38 □ exit a forum of target →
39     state := thinking;
40     target := ⊥;
41     priority := ⟨i, ∞⟩; /* reset the priority to a minimal value */
42     for each msg ∈ message_queue do send msg to pi+1;
43     message_queue := ∅;
44 ]

```

Fig. 4 Algorithm CTP-Ring2 for philosopher p_i

to p_{i+1} . Suppose that no other philosopher is interested in a forum, then p_j can attend Y after its confirmation message is returned (and while p_i 's request is still in transit). Since at this moment p_j has no knowledge of p_i 's sequence number sn , p_j could then issue another request for a forum, this time say Z, with the priority $\langle j, sn' + 1 \rangle$. Obviously, $sn' + 1$ could still be less than sn and, so, p_j is allowed to attend Z as it still has a priority higher than p_i does. Likewise, p_j can alternately establish Y and Z until its sequence number is finally larger than sn so that p_i 's next request will be detained by p_i until p_i has attended X. So,

with respect to this p_j , the number of 'forum switches' p_i has been bypassed is decided by the difference between sn and sn' . It can be seen that in the worst case, when p_i possesses sequence number sn , the other $N - 1$ philosophers could have sequence numbers equal to $sn - (N - 1)$, $sn - (N - 2)$, ..., and $sn - 1$, thereby yielding the context-switch complexity to $O(N^2)$.

Note that the above scenario cannot happen if messages are to be delivered in the order sent. This is because p_j 's request $Req(\langle j, sn' \rangle, Y)$ will not return to p_j until p_i 's request $Req(\langle i, sn \rangle, X)$ has arrived at p_j . Therefore, before

p_j issues another request, it must have learned of p_i 's sequence number sn , and so its next request must have a priority lower than $\langle i, sn \rangle$. Hence, the context-switch complexity will be $O(N)$.

Even if messages are not guaranteed to be delivered in FIFO order, we can still improve the context-switch complexity to $O(N)$ by slightly modifying the algorithm to let a philosopher p_i actively learn of another philosopher's sequence number via the messages circulating between them. To do so, request messages now have this form:

- $Req(\langle i, sn \rangle, sn_{max}, X)$

where the extra sequence number sn_{max} carried by the request is the maximum sequence number of the philosophers through which the request has passed. Initially, sn_{max} equals p_i 's SN when p_i issues the request (and note that sn is also equal to p_i 's SN). Upon receiving the request, a philosopher p_j adjusts its own SN to be the maximum value of its current SN and sn_{max} . When p_j is to forward the request to its successor, it adjusts sn_{max} to be the maximum value of its current SN and sn_{max} . (Note that if the request is to be detained by p_j , then in between the time p_j receives the request and the time it releases the request, p_j 's sequence number SN may have been adjusted a number of times.)

An extra sequence number is also added to confirmation messages so that they now have these two forms:

- $Conf/Capt(\langle i, sn \rangle, sn_{max}, X)$;
- $Conf(i, \langle j, sn \rangle, sn_{max}, X)$.

The extra parameter sn_{max} is initialised and used similarly to the new request messages.

We shall refer to this modified algorithm as CTP-Ring3. It is clear that the modification does not affect the mutual exclusion, bounded delay, and concurrent entering properties.

The simulation results for both CTP-Ring2 and CTP-Ring3 are also given in the tables. We can see that the performance of the system is improved significantly. For example, in Tables 2–3 when $m=3$, the average context switches per request is approximately 2 for both CTP-Ring2 and CTP-Ring3, as opposed to 19.08 for CTP-Ring1. Moreover, on average, up to 12–13 philosophers can be in the meeting room simultaneously per round for CTP-Ring2 and CTP-Ring3, as opposed to 1.49 for CTP-Ring1.

Moreover, we have also conducted a simulation for CTP-Ring3 where request messages may be deleted before they return to their originators. Recall from the algorithm that after a philosopher p_i has issued a request, if another philosopher captures p_i before the request returns to p_i , then p_i will immediately issue a confirmation message (of the form $Conf$). So p_i 's request becomes obsolete and can be deleted right away. So we can modified the algorithm such that if a philosopher p_j holding a request by p_i receives a confirmation message initiated by p_i , then p_j can delete p_i 's request immediately. In addition, since messages are not guaranteed to be delivered in the order sent, it is also possible that p_j may receive a new request by p_i while it is holding an old request by p_i . For example, suppose p_i has sent out a request to p_j but the request has not yet arrived at p_j . Subsequently, p_i is captured to send a confirmation message to p_j . Suppose further that the confirmation message reaches p_j and then returns to p_i before p_i 's request arrives at p_j . After p_i has attended a forum it may issue a new request to p_j . If p_j receives the new request while it is holding p_i 's old request, then obviously the old request can be deleted as well.

We refer to the modified algorithm as CTP-Ring4. On average, a request will travel about half of the ring before it is deleted. So the message complexity can be reduced to approximately $1.5N$. The simulation results for CTP-Ring4 are given in final Tables 4 and 8. Note that the message complexity remains $2N$ when $m=1$. This is because in this case no request will be detained, and so it is not possible for a philosopher to receive a confirmation message initiated by p_i while holding p_i 's request. On the other hand, when $m=N^*$, no philosopher will ever be captured and so the message complexity remains at $2N$.

5 Conclusions and future work

We have presented two distributed algorithms CTP-Ring1 and CTP-Ring2 (and its modifications CTP-Ring3 and CTP-Ring4) for CTP on a ring network where each philosopher can only communicate directly with its two neighbouring philosophers. Although CTP-Ring2 requires N more messages per entry to the meeting room, its dynamic performance is improved significantly over CTP-Ring1.

Due to the structure of rings, it is easy to see that an algorithm for CTP requires, in the worst case, at least N messages per entry to the meeting room. Thus, CTP-Ring1 achieves an optimal message complexity. Note that only one message is initiated by a philosopher requesting a forum; the ring structure renders the other $N-1$ messages to be generated by the other philosophers for the request to be fully acknowledged. As we can see, the single message initiated by the requesting philosopher is not 'enough' to boost the system's performance if philosophers request fora concurrently in such a way that neighbouring philosophers are interested in different fora (which, as can be seen, is likely to occur when contention for the meeting room is high). Therefore, in CTP-Ring2 and its modifications, a second message is initiated by the requesting philosopher to collect more information about the other philosophers' states and possibly to capture philosophers having the same interest to enter the meeting room. However, this second message also increases the algorithm's message complexity to $2N$. We conjecture that such a message is needed for the algorithm to overcome the poor performance exhibited by CTP-Ring1. A more formal treatment of this issue will be carried out in future work.

Finally, unlike Ricart and Agrawala's message-passing algorithm for N -process mutual exclusion [6], the sequence numbers used by the algorithms presented in the paper cannot be bounded within the range x to $x+N-1$. To see this, for example, consider CTP-Ring1. Suppose that p_i has initiated a request for X with a sequence number n . If no philosopher is interested in a different forum, then while p_i stays in the meeting room, another philosopher p_j could make an arbitrary number of entries to the meeting room to attend X (because no philosopher will detain p_j 's requests). This would then cause p_j 's SN to be adjusted to an arbitrary value greater than n possessed by p_i in its priority $\langle i, n \rangle$. Another direction for future work is therefore to attempt to bound the range of sequence numbers, or disprove such a possibility.

6 Acknowledgments

Research supported in part by the National Science Council, Taipei, Taiwan, under Grants NSC 85-2213-E-002-059 and NSC 86-2213-E-002-053. We thank the anonymous referees for useful comments on the paper.

7 References

- 1 JOUNG, Y.-J.: 'Asynchronous group mutual exclusion (extended abstract)', Proceedings of 17th ACM Symposium on *Principles of Distributed Computing* (PODC), pp. 51-60, June 1998
- 2 JOUNG, Y.-J.: 'The congenial talking philosophers problem in computer networks', Technical report, Dept. of Info. Management, National Taiwan University, Taipei, Taiwan, 1998. Extended abstract appeared in Proceedings 13th International Symposium on *Distributed Computing* (DISC99)
- 3 LAMPORT, L.: 'Time, clocks and the ordering of events in a distributed system', *Commun. ACM*, 1978, **21**, (7), pp. 558-565
- 4 Java applets, animating the algorithms in the paper: <http://jyoung.im.ntu.edu.tw/congenial/>
- 5 HOARE, C.A.R.: 'Communicating sequential processes', *Commun. ACM*, 1978, **21**, (8), pp. 666-677
- 6 RICART, G., and AGRAWALA, A.K.: 'An optimal algorithm for mutual exclusion in computer networks, 1981, **24**, (1), pp. 9-17