JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **27**, 201–204 (1995)

On the Correctness of Goscinski's Algorithm[1]

ROBERTO BALDONI,[2] BRUNO CICIANI, AND GIACOMO CIOFFI

Dipartimento di Informatica e Sistemistica, University of Rome "La Sapienza," via Salaria 113, I-00198 Rome, Italy

In this paper, the correctness of the mutual exclusion algorithm proposed by Goscinski (*J. Parallel Distribut. Comput.* 9(7), 77–82 (1990)), hereafter $\mathscr{G}$, is discussed and its features are compared with other token-based algorithms already published. In particular, we show that $\mathscr{G}$ works correctly only using a communication system that guarantees a total ordering of messages, otherwise it is incorrect. We further give a modified version of $\mathscr{G}$, hereafter $\mathscr{BCC}$, and show that $\mathscr{BCC}$ is actually a simple modification of the Suzuki–Kasami algorithm (*ACM Trans. Comput. Systems* 3(5), 344–349 (1985)). © 1995 Academic Press, Inc.

1. INTRODUCTION

Many distributed mutual exclusion (dmutex) algorithms have been proposed in the literature and they have been classified by Raynal [8] in two families: the *permission-based* algorithms and the *token-based* ones. The token-based algorithm family exchanges a virtual object among the processes called token, and its possession gives a process the permission to enter its critical section (*CS*). We can split the token-based algorithms in two subfamilies (in function of how the token is exchanged among the processes): the *token-asking* [2, 12, 10] and the *logical ring* [7]. In token-asking algorithms, the token is passed among the processes only if someone asks for it.

Dmutex algorithms are based on some communication assumptions. A communication system usually guarantees one of the following message ordering abstractions:

No ordering (*NO*): (a) the communication is reliable and (b) the transmission times are unpredictable but finite;

Partial ordering (*PO*): (a), (b), and (c) for each pair of processes messages are received in the order they were sent;

Causal ordering (*CO*): (a), (b), and (d) if two emissions of messages are ordered by the "happened before" relation [6], each process receives such messages in that order [9];

Total ordering (*TO*): (a), (b), and (e) all the messages are fully ordered among themselves. (This abstraction is also known as logically instantaneous communication mode.)

Of course (e) implies (d), which in turn implies (c). Such relation means that a TO system needs more complex protocols to be implemented than a CO one and so on. Therefore, the message ordering abstraction that the system must guarantee, in order that the algorithm works correctly, is an important comparison index.

Due to the relative simplicity of the protocol (only two causal messages) and to the centralized approach (there is only one data structure that contains requests to be served), token-based algorithms need relaxed assumptions than the others. In particular, while the permission-based algorithms [11, 1] require a PO system to work properly, token-based algorithms [2, 12, 10] need only an NO system.

$\mathscr{G}$ falls into the token-asking family, in particular it is an all-asking such as [12, 10] and, therefore, the number of messages exchanged per *CS* is *n* where *n* is the number of processes; $\mathscr{G}$ is based on two communication system assumptions: (i) there is a homogeneous or heterogeneous network and (ii) it is an error-free network; i.e., the messages are not lost and are delivered in the order they were sent. The latter assumption is ambiguous and it can be interpreted in two ways: $\mathscr{G}$ runs on a PO system or on a TO system. However, in both cases, $\mathscr{G}$ needs a complex communication system than other similar token-based algorithms.

In the sequel we will show that, using a PO system, $\mathscr{G}$ is incorrect; i.e., it prevents neither the occurrence of starvation nor deadlock. Starvation may occur because the algorithm may lose requests and deadlock may occur because the algorithm may duplicate requests. We will also show that $\mathscr{G}$ runs only on a TO system, while with a CO system starvation may yet manifest itself. In the final section, we will make the necessary corrections to $\mathscr{G}$ in order to obtain an algorithm ($\mathscr{BCC}$) that runs on an NO system. We will see that $\mathscr{BCC}$ is actually a simple modification of [12]. For brevity's sake all formal proofs are available in [3].

Finally, we want to underline that the worth of $\mathscr{G}$ was to point out the necessity of introducing priority-based discipline in dmutex algorithms. Indeed, all the proposed algorithms in the literature used FirstCome-FirstServed as the discipline to serialize requests. A thorough examination of the insertion impact of the priority in dmutex algorithms is in [2].

2. THE $\mathscr{G}$ ALGORITHM

Two algorithms were proposed in [5]: one is suitable for the environment requiring priorities (P-system) and the

[1] Work partially supported by the Consiglio Nazionale delle Ricerche under Contract 92.02294.CT12.

[2] E-mail: baldoni@dis.uniroma1.it.

other one for real-time environments (RT-system). The differences between these algorithms are implied by the features of environments as well as methods used to ensure freedom from starvation. The skeleton of these two algorithms is the same, therefore in the following we will describe only the P-system version. The algorithm is based on: (i) one process per site, (ii) no knowledge about the token location. $\mathcal{G}$ uses in each process a local priority queue (Q) to store incoming requests from other processes and a priority queue associated with the token (P) to memorize the processes waiting for *CS* entry (for further details, see [5]). The structure of the code executed by process *i* is the following (the notation is similar of $\mathcal{G}$):

```
    procedure wants_to_enter (p); {p is the
      priority of the request}
    begin
         requesting := true;
         if have_token
            then begin
                 broadcast REQUEST(i, p(i));
1.               wait until receive TOKEN(P)   ;
2.               have_token := true;
            end;

    critical section;

         if ¬((Q = ∅) ∧ (P = ∅))
            then begin
3.               append(P, Q) → P;
4.               have_token := false;
5.               send TOKEN(tail(P)) to head(P);
            end;
         requesting := false;
    end.

    procedure request_arrives (j, p); {this procedure
must be done atomically}
    begin
6.       if ¬ have_token
            then discard the request;
            else begin
                  add (j, Q) → Q;
7.                acts like 3 and 5;
            end
    end.
```

2.1. Correctness

A *token-asking* algorithm must avoid loss and multiplication of requests. The loss is due to the lack of processes

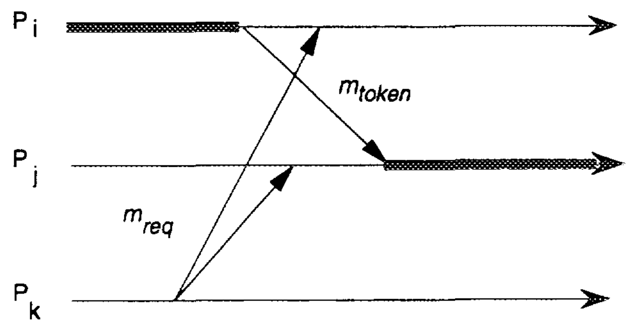

FIG. 1. Loss of a request.

that register requests, the multiplication is due to multiple insertion of the same request in *P*. The loss may lead to *starvation* since a process may wait forever the token (step 1), whereas the multiplication leads to token loss since after a finite time the token will be passed (step 5 or 7) to a process not waiting for it. We consider this event as a *deadlock*. In the sequel we will consider message duplication only, since the removal of duplicated requests executed by every process carries to multiplication removal.

Suppose that the communication system guarantees PO. In this hypothesis $\mathcal{G}$ creates the conditions for starvation and deadlock.

Starvation may occur since requests can be lost. Indeed, according to $\mathcal{G}$, the process having the token disables the reception of requests (step 4) before executing the send-token statement (step 5). To the contrary, a process starts to record requests only upon the receipt of the token (step 2), otherwise it discards requests (step 6). A PO system may deliver a request both to the process sending and to the process receiving the token when they are disabled to accept the request (see Fig. 1). Note that a time-out based mechanism to retransmit requests (as suggested in [5]) does not avoid starvation, since a request could be always delivered in time intervals during which no process accepts requests (see Fig. 2).

Deadlock can occur since a request can be delivered, by a PO system, both to the process sending and to the process receiving the token when they are enabled to accept the request (see Fig. 3) with consequent creation of an unexpected request (i.e., requests in *P* without a process waiting for them). Indeed, in $\mathcal{G}$ no check is done to avoid the presence in *P* of multiple copies of the same request (step

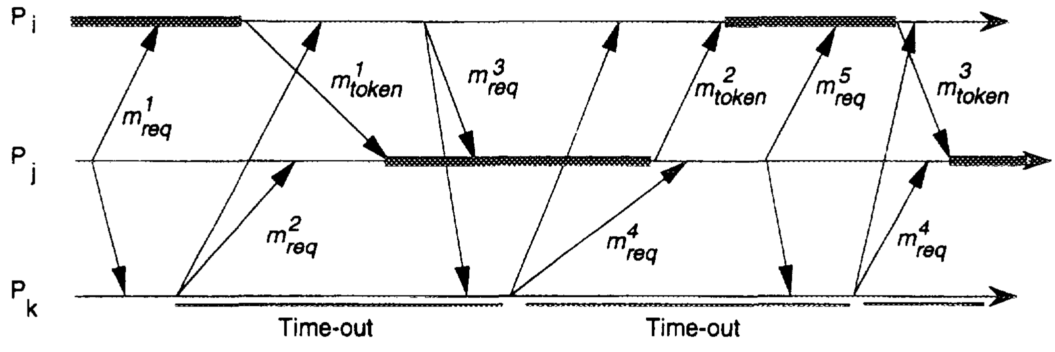

FIG. 2. Starvation due to infinite losses of a request.

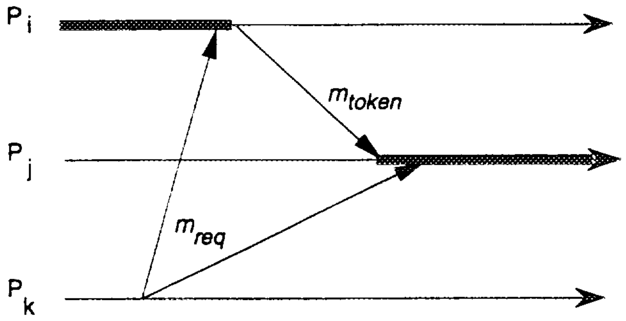

FIG. 3. Request duplication.

3 and 7). Other token-asking algorithms [2, 12] mark requests with a sequence number. This label avoids having a request served more than once.

Let us suppose to have a CO system. In this case the duplication of requests is removed tout-court, since the broadcast message m_{req} of Fig. 3 is causally ordered with m_{token}. Therefore, the communication system will deliver m_{token} to the user process only after the delivery of m_{req}, even though the communication system will receive m_{token} first. Unfortunately the messages of Fig. 1 are not causally ordered, therefore starvation may yet manifest itself.

If we assume a TO system, we avoid both the loss of requests and the request multiplication: the duplication is avoided since total order implies causal order, the loss is avoided since the situation of Fig. 1 can never occur. We can get a total order attaching a progressive number, given by a central coordinator, to each message and the system will deliver the messages according to that order to each process. Such ordering can be easily implemented making simple modifications to the centralized broadcast protocol proposed in [4].

3. THE $\mathscr{BCC}$ ALGORITHM

Considering the algorithm of Section 2, the filter of step 6 plays a key role in avoiding request loss, whereas the data structures Q and P play a key role in removing request duplication. In particular, to avoid any loss of requests, we

```
     procedure wants_to_enter ($p$); {$p$ is the priority of the request}
     begin
          requesting := true;
          if have_token
             then begin
                  $RN[i,1] := RN[i,1] + 1$ ; $RN[i,2] := p$ ;
                  broadcast REQUEST($i$, $RN[i,1]$, $RN[i,2]$);
                  wait until receive TOKEN($P$, $LN$) ;
                  have_token := true;
             end;

     critical section;

1.        $LN[i] := RN[i,1]$ ;
2.        for each $j \in \{1,\dots n\} - \{i\}$ do
3.           if $(\neg in(P,j) \wedge (RN[j,1] = LN[j] + 1))$ then add($j$, $Q$) → $Q$;
4.        append($P$, $Q$) → $P$;
          if $\neg(P = \emptyset)$
             then begin
5.                have_token := false;
6.                send TOKEN(tail($P$), $LN$) to head($P$);
             end;
          requesting := false;
     end.

     procedure request_arrives ($j,n,p$); {this procedure must be done atomically}
     begin
7.        if $RN[j,1] < n$
             then begin
                  $RN[j,1] := n$ ; $RN[j,2] := p$ ;
                  if have_token $\wedge$ ($\neg$requesting) $\wedge$ $(RN[j,1] = LN[j] + 1)$
                     then begin
8.                        have_token := false;
9.                        send TOKEN(tail($P$), $LN$) to head($P$);
                     end
             end
          else discard the request;
     end.
```

FIG. 4. The $\mathscr{BCC}$ algorithm.

```
if $(P = \emptyset)$
  then for each $j \in \{1, \ldots n\} - \{i\}$ do
    if $(\neg \text{in}(P, j) \wedge (RN[j, 1] = LN[j] + 1))$ then add$(j, P) \rightarrow P$;
```

FIG. 5. The $\mathcal{BCC}$ algorithm using an FCFS discipline.

should prevent processes from discarding requests or that they at least discard only requests that were already served. Therefore, step 6 should be removed or at least changed. The request duplication can be dealt with by increasing the local and global information about the requests. P and Q does not contain indeed enough information.

In this section, we correct $\mathcal{G}$ in order to have an algorithm that (i) runs on an NO system and (ii) serves requests according to their priority avoiding starvation. The $\mathcal{BCC}$ algorithm is shown in Fig. 4.

Duplication is avoided by applying the sequence number mechanism cited in the previous section (for further details, see [12]). We need a new data structure that travels with the token LN and a local one at each process RN. LN is a vector whose size is n and $LN[i]$ stores the sequence number of the last served request of process i. RN is an array with n rows and 2 columns. Process i stores in $RN_i[j, 1]$ the largest sequence number received from process j and in $RN_i[j, 2]$ its priority.

When process i exits from the CS, it updates its sequence number in the LN vector (step 1) and appends to P the requests stored in RN_i verifying the condition of step 3. Such a condition guarantees that P never include requests duplications. Due to the simplicity of the protocol (only two causal messages: REQUEST and TOKEN), we note that requests whose sequence number is less than that stored in RN can be discarded since they have already been served. Hence the filter of step 7.

From the code of Fig. 4, we see that an FCFS discipline can be obtained by replacing lines 2 to 4 with those shown in Fig. 5. Doing so, we get the Suzuki–Kasami algorithm. Hence, the proofs that $\mathcal{BCC}$ guarantees mutual exclusion and is deadlock free are omitted since they are similar to those in [12]. As far as starvation is concerned, the proof given in [5] holds.

4. CONCLUSIONS

In this paper, we showed that the dmutex algorithm proposed by Goscinski [5] runs only on a TO communication system. If an NO communication system, which is the standard system of other similar token-based algorithms [2, 10, 12], is available, $\mathcal{G}$ prevents neither starvation nor deadlock. We gave a correct version of the algorithm ($\mathcal{BCC}$) and we showed that $\mathcal{BCC}$ is actually a simple modification of the Suzuki–Kasami algorithm [12]. Finally, we want to emphasize that Goscinski was the first to point out the necessity of introducing a priority-based discipline in dmutex algorithms. A thorough examination of the insertion impact of the priority in dmutex algorithms can be found in [2]. All the formal proofs concerning the incorrectness of $\mathcal{G}$ and the correctness of $\mathcal{BCC}$ can be found in [3].

Received July 30, 1992; revised April 22, 1994; accepted May 18, 1994

REFERENCES

1. R. Baldoni, An $O(n^{m/(m-1)})$ distributed algorithm for the k-out of m-resources allocation problem. *Proceedings of the 14th conference on Distributed Computing Systems*, pp. 81–88. IEEE Press, New York, 1994.
2. R. Baldoni and B. Ciciani, Distributed mutual exclusion algorithms with priority. *Inform. Process. Lett.* **50,** 165–172 (1994).
3. R. Baldoni, B. Ciciani, and G. Cioffi, *Token-Asking distributed mutual exclusion algorithm with priority*. Technical Report RAP. 05.94, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Jan. 1994.
4. J. Chang and N. F. Maxemchuk, Reliable broadcast protocol. *ACM Trans. Comput. Systems* **2**(3), 251–273 (1984).
5. A. Goscinski, Two algorithms for mutual exclusion in real-time distributed computer networks. *J. Parallel Distribut. Comput.* **9**(7), 77–82 (1990).
6. L. Lamport, Time, clocks and the ordering of events in a distributed systems. *Comm. ACM* **21**(7), 558–565 (1978).
7. G. Le Lann, Distributed systems: towards a formal approach. *Proceedings of the IFIP Congress*, pp. 632–646. North-Holland, 1977.
8. M. Raynal, Simple taxonomy for distributed mutual exclusion algorithms. *ACM Oper. Systems Rev.* **25**(1), 189–193 (1990).
9. M. Raynal, A. Schiper, and S. Toueg, The causal ordering abstraction and a simple way to implement it. *Inform. Process. Lett.* **39,** 343–350 (1991).
10. G. Ricart and A. K. Agrawala, Authors response to: On mutual exclusion in computer networks. *Comm. ACM* **26**(1), 147–148 (1983).
11. B. A. Sanders, The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Systems* **5**(3), 284–299 (1987).
12. I. Suzuki and T. Kassami, A distributed mutual exclusion algorithm. *ACM Trans. Comput. Systems* **3**(5), 344–349 (1985).

ROBERTO BALDONI was born in Rome on 1965. He received the "Laurea" in electronic engineering in 1990 and the Ph.D. in computer science in 1994 from the University of Rome "La Sapienza." Presently he is a post-doctoral fellow at the Dipartimento di Informatica e Sistemistica of the University of Rome "La Sapienza." His main research interests include operating systems, distributed algorithms, and performance evaluation of concurrency control algorithms in multidatabase systems.

BRUNO CICIANI received the "Laurea" in electronic engineering in 1980 from the University of Rome "La Sapienza." He is Professor of Computer Science at the University of Rome "La Sapienza." His current research activities include distributed computer systems, fault-tolerant computing, languages for parallel processing, and computer system performance and reliability evaluation. He has published many papers in international journals, and he is the author of the book *Manufacturing Yield Evaluation of VLSI/WSI Systems* to be published by the IEEE Computer Society Press.

GIACOMO CIOFFI received the "Laurea" in electrical engineering in 1961 from the University of Naples and "libera docenza" in electronic computers in 1970. He is Professor of Computer Science at the University of Rome "La Sapienza," where he teaches courses in logical design, computer architectures, and distributed computing. His research activities are focused on distributed systems and programming, parallel algorithms, VLSI architectures, and logical CAD. He has published many papers in international journals including *IEEE Trans. on Computers* and *IEEE Transactions on Industrial Control*.