A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network

J. M. HELARY, N. PLOUZEAU AND M. RAYNAL*
IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France

A distributed algorithm for mutual exclusion is presented. No particular assumptions on the network topology are required, except connectivity; the communication graph may be arbitrary. The processes communicate by using messages only and there is no global controller. Furthermore, no process needs to know or learn the global network topology. In that sense, the algorithm is more general than the mutual exclusion algorithms which make use of an a* priori *knowledge of the network topology (for example either ring or complete network). A proof of the correctness of the algorithm is provided. The algorithm's complexity is examined by evaluating the number of messages required for the mutual exclusion protocol.

Received June 1986, revised February 1987

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

1. INTRODUCTION

The mutual exclusion problem consists in ensuring that at a given time a logically or physically shared object is accessed by one process at most. This fundamental problem has to be solved when writing parallel or distributed systems. As an example of a distributed mutual exclusion problem, consider a distributed system made of several processes, each process monitoring some machine. From time to time processes need to ask a question of a console operator (for instance upon machine failure). Once a question from process P has been printed, P has to wait for an answer, which has to be given after some delay. While P waits, no other process is allowed to ask its own question: access to the console is protected by a critical section. When P's question has been answered, access to console is allowed to the other processes (P leaves the critical section). Such a control scheme frequently occurs in industrial control applications.

Since 1965 numerous algorithms have been designed either for centralised systems (i.e. systems with a shared memory accessed by several processes) or for distributed systems (i.e. systems where processes interact by the only means of message transmission). Lamport proposed in his paper on time and logical clocks, as a use of these devices, an algorithm for mutual exclusion in a distributed context,[5] based on request-queue replication and event ordering. Several algorithms reducing the maximal count of the messages involved in the mutual exclusion protocol were given later. For instance, Ricart and Agrawala's algorithm[9] needs $2(n-1)$ messages to perform its task, while Carvalho and Roucairol's uses between 0 and $2(n-1)$ messages;[1] a second algorithm from Ricart and Agrawala needs either 0 or n messages[10] (interested readers can find a presentation of the mutual exclusion problem and a survey on existing mutual exclusion algorithms in Ref. 8).

As far as the authors know, all the algorithms for mutual exclusion which are based on interprocess communication by messages make use of an *a priori* known topology (either complete or ring network). We shall focus in this paper on mutual exclusion in a network with an *a priori* unknown topology.

* To whom correspondence should be addressed.

Section 2 reveals the required system properties, Section 3 presents the main principles of the algorithm; while the algorithm itself if given in Section 4. Section 5 proves some fundamental properties and Section 6 evaluates the efficiency of the algorithm. Section 7 examines the effects of failures.

2. ASSUMPTIONS

2.1 Required network properties

A process is a sequential activity which interacts with other processes only by sending and receiving messages. Two processes are said to be neighbours if and only if they are connected by a direct communication line. Between two neighbour processes we assume there exists exactly one direct communication line. Every interprocess communication line is bidirectional and is endowed with the following behavioural properties: no loss of messages; no message alteration; finite transmission delay. Messages do not need to be delivered in the order they were sent. No assumption is made about the network topology, except connectivity.

2.2 Local process knowledge

The only knowledge owned by a process is local – the name of its neighbours: the global network structure as well as the total number of processes belonging to it will remain unknown to any process.

We assume that a process and its neighbours have distinct names (global properties of the algorithm, such as fairness, will be established by assuming that the names of all the processes are mutually distinct, but this assumption is not used in the definition of a process; in that sense, the algorithm ensures a totally distributed control).

3. PRINCIPLES OF THE ALGORITHM

The main feature of the algorithm is the simultaneous use of well-known distributed algorithmic techniques. Furthermore, the knowledge-transfer control technique has been added, in order to reduce the number of messages required by the message-routeing protocol.[4]

(i) Every request sent by a process is propagated in the network with a *flooding broadcast* technique (also called *wave* technique).[12] Every time a process receives a request from one of its neighbours it propagates it to its other neighbours, thus broadcasting the request message.

(ii) The knowledge-transfer control method improves this technique. Every message sent down through the network carries a control part made of process names. Upon receiving a request message a process uses the control part to learn a subset S of the processes which have received the same request (or are about to receive, depending on the different message speeds). Thus a process computes the subset T of its neighbours not belonging to S, adds T to S and sends the message (whose control part has been updated with S) to the processes in T.

(iii) Owning a special message, called the *token*, is a necessary condition to have the privilege to access the critical section. At any time there exists at most one token in the system (a similar approach can be found in Refs 6 and 10, where the token travels along a ring and a complete network respectively).

(iv) Granting the privilege is performed by a single process, which is the current owner of the token. This process chooses the next token owner and sends it the token.

(v) The set of all pending requests is fully ordered, with a strict order relation. This allows the algorithm to be deadlock- and starvation-free. To this end we use Lamport's technique,[5] based on logical clocks and message stamps, to set up a total ordering on the requests set. Requests with identical stamps will be distinguished by the mean of the names of the processes which created them. This is why the process names must be all different (see Section 2.2).

(vi) The path followed by a request from a requesting process to the token owner is marked off; the token uses that path in the opposite direction to reach the requesting process whose request is the next to be satisfied (as in the reflecting privilege technique in Ref. 7).

The principles of the algorithm can now be stated.

- When a process wants to enter the critical section it sends a request message to its neighbours and waits for the *token* message (see point i).
- Upon receiving a request message, a process P broadcasts that request to its neighbours not belonging to the control part of the message; this part is a subset of the set of process names carried by every request message (see point ii above). The request is added to P's set of known pending request.
- If the process P owns the token and is outside the critical section (see point iv) above), it extracts the oldest request R from its known requests set (see point v above). It then sends the token to the creator of R through N, the neighbour of P which sent him the request R (see point vi above). If process P is inside the critical section, it will behave as stated above upon exiting it.
- Upon receiving the *token* message, process P keeps it if the token's addressee is itself. Otherwise P hands it over to N, the neighbour which sent him the request being serviced (see point vi).
- A process can enter the critical section only if it owns the token.

4. THE ALGORITHM

4.1 Messages

Two kinds of message are sent between processes in the algorithm: *request* messages and *token* messages.

(α) A *request* message has the following structure: *req(req_id,rt_info)*. Messages of this kind are created and sent by any process which doesn't own the token and wants to enter the critical section. They are propagated by the other processes P_i as pointed out in Section 3 (points i and ii). The parameters have the following meaning.

- *req_id* unambiguously identifies the request in the whole system's history. The components of *req_id* are *req_origin* and *req_time*; the first one contains the request creator process name, the second one the logical clock value of this process at the request creation time.
- *rt_info* contains a request history, from the broadcasting point of view. The components of *rt_info* are *sender* and *already_seen*. The first one is the name of the sender of the message; the second one is a set of process names. Every process whose name is in *already_seen* has received or is about to receive the request. This set allows us to control the knowledge transfers between processes, thereby reducing the number of messages.

(β) A *token(lud,elec)* message has the following properties:

- At any time there exists at most one such message in the system.
- The last process which has received and kept the *token* message is said to own it. When the token owner sends the token away, no one will own it until it reaches its final addressee (which is the next owner); processes involved in this token transfer are said to handle it (they don't own it).
- To own the token is necessary to own the mutual exclusion privilege.

The *elec* component is the process name of the token final addressee. The *lud* component is an array whose ith subcomponent stores the value that P_i's logical clock had when P_i gave the privilege to another process (see point iv in Section 3). The *lud* array is read and modified under mutual exclusion, because it is carried by the token. Thus the only process which can access *lud* is the token owner or handler. Every subcomponent of *lud* is initialised with the value -1.

(γ) At initialisation, no message is present in the network.

4.2 Local variables of processes

The mutual exclusion algorithm used by the processes is implemented with an abstract object, which is distributed on every process. Every process P_i is endowed with local variables, locally implementing the abstract object. It can use four procedures:

- *enter_CS*
- *exit_CS*
- *receive_request*
- *receive_token*

These procedures are atomic except for the *wait*

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

instruction used in *enter_CS*. Process P_i uses the first and the second procedures when it wants to enter and leave the critical section, respectively. Upon receiving a request or a token message from the P_j part of the abstract object, process P_i uses the third or the fourth procedure, respectively. A fifth procedure *transmit_token* is internal to the abstract object; it is used by *exit_CS* and *receive_token*.

This abstract object can easily be built with Ada tasks. The variables which are local to P_i and implement the abstract object can be accessed (i.e. read or modified) by P_i only and within the four procedures we have just mentioned. These local variables are:

const

- *neighbours*$_i$.: set of process_id; initialised with the names of P_i's neighbours.
  var
- C_i: 0.. $+\infty$ **init** 0;
  This is P_i's logical clock.
- *token_here*$_i$: **boolean**;
  This boolean variable is true iff P_i has the token (as owner or handler). At initialisation time *token_here* is false for every process but one.
- *in_CS*$_i$: **boolean init false**;
  This variable is true iff P_i is inside the critical section; P_i is then the token owner (this implies *token_here*$_i$ = **true**).
- *req_array*$_i$: **array**[*neighbours*$_i$] **of list of** (*req_origin*,*req_time*) **init nil**;
  The list of request identifiers that P_i received from P_j is stored in *req_array*$_i$[*j*].

4.3 Algorithm for process Pi

The following notations are used in the text of the abstract object procedures. Symbols $\in$, $\oplus$, $-$ stand for the *element_of*, *append*, *delete* list operators, respectively. The set operator $-$ is also used. If X is a non-empty subset of a cartesian product $Y \times Z$ of totally ordered sets, the function *min*(X) we make use of in procedure *transmit_token* gives the couple $(y,z) \in X$ such that

$$\forall (y',z') \in X: (y < y') \vee (y = y' \wedge z < z')$$

Elements of X are thus totally ordered. Comments are introduced by $--$. Symbols $\wedge$ and $\neg$ stand for the **and** and **not** boolean operators respectively.

```
procedure enter_CS;
  begin
    if token_here_i
      then in_CS_i := true
      else
      --broadcast a request
      ∀k ∈ neighbours_i: send req((i,C_i),(i,neighbours_i ∪
          {i}))to P_k;
    endif;
    wait in_CS_i;
    --May be interrupted upon receiving a
        message.
  end enter_CS;

procedure exit_CS;
  begin
    in_CS_i := false;
    transmit_token;
  end exit_CS;

procedure receive_request(req((req_origin,req_time),
      (sender,already_seen)));
  begin
    if ∃(req_origin,t) such that(req_origin,t) ∈ req_array_i
        ∧ (t < req_time)
      then
      --Delete this old request
      req_array_i := req_array_i-(req_origin,t);
    endif;
    if (req_origin,req_time) ∉ req_array_i ∧ ¬
        (∃(req_origin,x) such that x > req_time)
      then
      --The request just received is a new
          one and is the youngest that P_1 ever
          received from process P_req_origin
      C_i := max(C_i,req_time) + 1;
      req_array_i[sender] := req_array_i[sender] ⊕
          (req_origin,req_time);
      --Broadcast the request
      ∀k ∈ neighbours_i - already_seen:
        send      req((req_origin,req_time),(i,already_
            seen ∪ neighbours_i)) to P_k;
      if token_here_i ∧ ¬ in_CS_i then transmit_token;
      endif;
    endif;
  end receive_request;

procedure receive_token(token(lud,elec));
  begin
    token_here_i := true;
      if elec = i
      then in_CS_i := true
      else
      --The token is following the path
          that the corresponding request
          established.
      let via be such that (elec,x) ∈ req_array_i[via];
      token_here_i := false;
      send (token(lud,elec)) to P_via;
    endif;
  end receive_token;

procedure transmit_token;
  begin
    --Compute the set X of the processes
        owning a pending request then find
        the oldest and send it the token.
    let X be {(orig,t) such that((orig,t) ∈ req_array_i
        ∧ (lud[orig] < t))};
    if X ≠ { }then
      (elec,x) := min (X);
      let via be such that (elec,x) ∈ req_array_i[via];
      req_array_i[via] := req_array_i[via] - (elec,x);
      lud[i] := C_i;C_i := C_i + 1;--Line ref-
          erenced (A) in the proof.
      token_here_i := false;
      send token(lud,elec) to P_via;
    endif;
  end transmit_token;
```

5. PROOFS

5.1 Mutual exclusion

Theorem 1: the algorithm ensures mutual exclusion

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

Proof

We have to prove that the number of processes which are inside the critical section is less than or equal to 1.

$$Card(\{P_i \textbf{ such that } in\_CS_i = true\}) \leqslant 1 \quad \textbf{(ME1)}$$

A process cannot enter the critical section if it doesn't own the token. Indeed, as stated by the text of the algorithm:

$$\forall i\text{: } in\_CS_i \Rightarrow token\_here_i \quad \textbf{(ME2)}$$

We show that the following predicate **ME3** is a system invariant:

$$Card(\{P_i \textbf{ such that } token\_here_i = true\}) \leqslant 1 \textbf{(ME3)}$$

(ME2) and **(ME3)** will prove **(ME1)** and hence Theorem 1.

Initially predicate **(ME3)** is true and no message exists in the network; there is one process P_j and only one such that $token\_here_j$ is true; no other process can send a token message. When P_j gives the token to another process P_k, the following statements are executed:

- In P_j (in procedure *transmit_token* or *receive_token*)
  - ○ $token\_here_j$:= **false**;
  - ○ **send** token(lud,elec) **to** P_k
- in P_k (upon the reception of the token)
  - ○ $token\_here_k$:= **true**

The variables *token_here* are modified in no other parts of the algorithm. Thus we conclude that predicate **(ME3)** is a system invariant. This proves Theorem 1.

5.2 Request routeing

Lemma 1

Let $R = (i, rt_i)$ be a request created and broadcast by process P_i. For all j different from i, if R reaches P_j then this occurs after a finite delay along an elementary network path. If R never reaches P_j then there exists a request $R' = (i, rt'_i)$ with $rt'_i > rt_i$, which has overtaken R while R and R' were moving in the network; hence R has been satisfied.

Proof

To prove Lemma 1 we will show that the following property is an invariant.

Property

If P_j receives a message $req((i,rt_i),(k,already\_seen))$ then $already\_seen = z_a \cup z_e$, where z_a is the set of the names of the processes on the path followed by request (i, rt_i) from P_i to P_k, and z_e is the set of the names of the neighbours of every process in z_a. Indeed, if P_j is a neighbour of P_i then we have (see procedure *enter_CS*): $(k = i) \wedge (already\_seen = neighbours_i \cup \{i\})$. Thus the property is verified in that case.

Now assume that the property is verified for P_k. The control information received by P_k is $already\_seen_k$. As stated by the procedure *receive_request*, the *already_seen* parameter of the message received by P_j is (cf. procedure *receive_request*):

$$already\_seen = already\_seen_k \cup neighbours_k$$

Thus l is an element of *already_seen* iff:

- there exists some P_m on the path followed by R from P_i to P_k such that $l = m$ or P_l is a neighbour of P_m (recurrence hypothesis).
- or $P_m = P_k$
- or P_m is a neighbour of P_k.

This proves the asserted property for P_j.

As a consequence of this property the path T followed by a request $R(i, rt_i)$ issued by P_i and reaching P_j is acyclic, because a process never forwards a request to another process which is on T or is a neighbour of a process on T. Paths followed by a request are thus of finite length. From this fact, together with hypotheses on the network (connectivity, finite transmission delays), and since there is no loop in the procedure *receive_request*, request R is propagated throughout the network, building up a spanning tree (with root P_i); every process will belong to this tree, unless a process P discards R, that is to say doesn't pass this request on to its neighbours not belonging to *already_seen* (if any). By the procedure *receive_request*, such a deletion will happen if and only if a younger request $R' = (i, rt'_i)$, (i.e. with $rt'_i > rt_i$), reached P before R did (R' can overtake R along communication lines); now this implies that P_i has been allowed to issue R', in other words that R was satisfied (a process is not allowed to issue a new request while its last one is not satisfied). **QED**.

5.3 Token routeing

Lemma 2

When a process P_i owning the token grants another process P_j the privilege to enter the critical section (and then P_j will own the token), i.e. when answering to a request $R = (j, rt_j)$, the token follows in the opposite direction the path built by R from P_j to P_i.

Proof

Let P_k be a process on the path γ_R used by R to go from P_j to P_i. Since P_k is involved in the broadcasting of R, P_k's local context upon receiving R is as follows: either no request created by P_j is in req_array_k or there is a request (j, rt'_j) with $rt'_j < rt_j$ (i.e. a request already satisfied). Thus process P_k stored R in $req\_array_k[l]$, where P_l is P_k's predecessor on path γ_R and R is the only request with origin P_j stored in req_array_k. Request R is kept in req_array_k until one of the following events occurs:

- P_k receives the token (lud, j) message
- P_k receives a $req(j, rt''_j)$ message, where $rt''_j > rt_j$

The second event cannot occur before the first one because P_j is not allowed to send a new request until it enters and exits the critical section. Moreover P_i transmits the token to process P_{via} which sent him request R; thus P_{via} is on path γ_R and it hands the token over to its predecessor on γ_R. This behaviour is the same for all P_k on γ_R, as stated by the text of procedure *receive_token*. **QED**.

From Lemmas 1 and 2 it follows:

Corollary

The path followed by the token from its sender P_i to its final addressee P_j is acyclic.

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

5.4 Absence of deadlocks

Lemma 3

If a request $R(i, rt_i)$ sent by P_i is not satisfied we have $lud[i] < rt_i$.

Proof

If process P_i never owned the token before it sends R then $lud[i] = -1 \wedge rt_i \geqslant 0$. If P_i owned the token at least once, it must have left it in order to send a new request. The statements at line (A) in procedure *transmit_token* imply that $rt_i \geqslant lud[i]+1$. **QED.**

Theorem 2: the algorithm is deadlock-free

Proof

Deadlock means that, whilst no process is in the critical section, one or several processes wish to enter it and no one will be allowed to do it within a finite delay. Thus, at least one pending request $R = (orig, rt)$ has been issued and is not satisfied; by Lemma 1 it will reach every other process within a finite delay, in particular the owner of the token, say P_i; P_i executes the procedure *transmit_token* either upon receiving R or upon exiting the critical section. If P_i decides not to give the token, for all requests (j, rt_j) stored in req_array_i we must have $lud[j] \geqslant rt_j$ (see procedure *transmit_token*). In particular, $lud[orig] \geqslant rt$; but, by Lemma 3, $(orig, rt)$ being not satisfied we have $rt < lud[orig]$. This contradiction shows that no process owning the token can keep it for ever if there exists a pending request. Finally, Lemma 2 shows that a token sent in the network will reach its addressee within a finite delay. **QED.**

5.5 Absence of starvation

Theorem 3: there is no starvation

Proof

It is assumed that every process having entered the critical section eventually executes the *exit_CS* procedure.

We have to prove that every request is satisfied within a finite delay. If a request R is never satisfied, Lemma 1 shows that every process in the system knows R within a finite delay after its creation. Upon receiving R, a process P_i updates its logical clock, whose value is then greater than R's stamp. Thus every request created by P_i after this update has stamps greater than R's. The number of requests satisfied before R is finite. Moreover, when the token is moving to satisfy a request R' this one is deleted from the request table *req_array* of every process on the path γ_R, followed by the token and the token transfer is done within a finite delay (Lemma 2 and Corollary). Thus request R will become the oldest within a finite delay and process P, which created R, will then be chosen as the new token addressee. Consequently a request R cannot be 'never satisfied'. **QED.**

6. MESSAGE TRAFFIC

A complexity measure of a distributed algorithm is the number of messages it needs to perform an *enter_CS* and *exit_CS* sequence. Moreover, if the maximal transmission delay Δ on a network line is known we can compute the maximal delay needed to perform that sequence. Four network shapes will be considered: the tree, ring, complete network and general case topologies.

Whatever topology we consider, there is no need to send any request message when the token's owner wants to enter its critical section. The required number of messages is then zero. This is not the case if process P_i wants to enter its critical section and doesn't own the token. The total number of messages sent is the sum of the number required to broadcast the request and the number required to move the token back. Let d be the diameter of the network; broadcasting a request to every process or moving the token from the current owner to the new one takes at most $d\Delta$ units of time. Thus the total transmission time of a complete request operation takes at most $2d\Delta$ units of time. The longest acyclic path in a graph of n vertices has a length of $n-1$. This is also the longest path that the token may follow to find its new owner. Therefore moving the token requires at most $n-1$ messages. We will now consider four particular network topologies.

Tree topology

Broadcasting a request requires exactly $n-1$ messages. Moving the token takes between 1 message (if the sender and the addressee are neighbours) and d messages, where the diameter d is here the length of the longest path ($1 \leqslant d \leqslant n-1$). Thus the total number of messages varies from n to $n-1+d$. A particular case is the **line topology**, where $d = n-1$: bounds are then n and $2(n-1)$.

Ring topology

Broadcasting a request requires at least $n-1$ messages and at most $n+1$ messages. Thus the total number of messages varies from n to $2n$. When the algorithm is used on a ring topology, its behaviour is similar to the second algorithm presented in Ref. 7 (the one called *reflecting privilege* algorithm). Whatever the relative locations of the token owner and the token addressee, the token moves along a path that the request followed in the opposite direction.

Complete network

The knowledge transfer control principle allows us to use exactly $n-1$ messages to broadcast a request. Moving the token requires a single message. Thus the total message number is n. If the complete network topology is *a priori* known by all the processes, we can simplify the algorithm and obtain a new one whose behaviour is similar to the one of Ricart and Agrawala's algorithm.[10] We use logical clocks instead of request counters and we send the token to the creator of the oldest request known instead of using a logical ring built on the set of the processes waiting for the token (these two techniques are two different ways of ensuring the fairness property).

General case

A request message can be sent at most twice on a given transmission line (once in each direction) which connects

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

two neighbours. The total request message number lies thus between $n-1$ and $2e$, where e is the number of edges ($e \leqslant n(n-1)/2$); let us point out that this theoretical upper bound is often over-evaluated, since the control knowledge transfer technique reduces the actual number of messages in a wave-broadcasting protocol according to the density of the graph.[4] On the other hand, moving the token requires at least 1 message and at most $n-1$ messages. Thus the total message number is at least n and at most $2e+n-1$.

7. ENHANCEMENTS

Although fault-tolerance is not the main topic of this paper, we examine in this section effects of failures on the proposed algorithm and consider how assumptions about the network behaviour may be violated in a real distributed system. We assumed that no transmission line can lose or alter messages. But, although a real (i.e. physical) channel cannot own these properties, it is easy to achieve such a high transmission quality by using protocols for error- and loss-free transmission. Such protocols are known;[13,14] they are implemented at the transport level in the network. Our algorithm doesn't require message-order preservation.

7.2 In case of failure

We now consider a system where messages are safely transmitted on communication channels. These channels may fail at any time: if this occurs while some message is being transmitted, the message is considered not to be sent by the transport level of the underlying network, and its sender is notified that its message could not be sent because of channel failure.

7.1.1 Channel failure

We consider one particular mutual exclusion request, say from P_i, and we assume that some channel C_{jk} connecting P_j to P_k goes down; then several situations may occur.

(1) If channel C_{jk} is a minimal cut then deleting C_{jk} breaks network connectivity: in that case we have two separate sub-graphs (we recall that the communication graph is an unoriented one: strong connectivity is equivalent to connectivity). The token is in one of these sub-graphs: there is no violation of the mutex invariant because there is at most one token in the system. Completion of request broadcasts is not possible, nor token routeing if the token addressee is in the other sub-graph. Some action has to be performed upon recovery of C_{jk}, in order that pending requests broadcasts flood the other part (which was unreachable before) and that pending requests complete. This may be achieved by restarting a broadcast for every request which was pending at the time of the failure using the original timestamp of that request (this gives it a high priority compared to requests submitted after the failure). Thus every neighbour P_j of a recovering channel C_{jk} or C_{kj} sends to process P_k every pending request in *req_array*$_j$. Note that if some process P had a pending request then it couldn't broadcast another request: if a failure postpones completion of a request then the requester implicitly waits for recovery.

(2) If suppression of C_{jk} doesn't split the graph then we have the following situation. Broadcasts complete normally (because the network is still strongly connected). When the token is travelling to its final addressee P_i, it follows a spanning tree branch to the root. If C_{jk} wasn't on this path, then its failure doesn't prevent the token from reaching the root. If C_{jk} was on the path from token to P_i at the time of failure then the token cannot go through C_{jk} and reach P_i. It is possible to strengthen the routeing protocol if several different exits to the root are computed to every node: in case of failure of one exit, another may be available. We modify the algorithm as follows: during the request broadcast phase, every process P_i forwarding a request to its neighbours not in *already_seen* gives them the identities of P_i's brothers in the request spanning tree, as well as the identities of P_i's children (in order that every child knows the identities of its brothers). Upon receipt of a request message from P_i, process P_j learns several items of information: identity of its father in the spanning tree; identity of its uncles; identity of its brothers. Its parents are its father and every process which is in the intersection of its uncles' set and neighbours' set. During the token routeing phase, P_j has to transmit the token to one of its reachable parents in the spanning tree. This enhancement makes use of the possibility of more than one path from every node to the root of the spanning tree; at every node there may exist more than one exit for the token. It is still possible that no exit towards the root P_x is available at some node (even if the failed channel is not a minimal cut): in that case, this node has to compute a new addressee using procedure *transmit_token* and send the token to the new root. The interrupted request from P_x is still pending but won't be forgotten (because only a new request from P_x can erase the spanning tree built by the previous one): the new addressee will take it into account upon releasing the mutual exclusion and choose P_x as next token addressee.

7.1.2 Process failure

We assume that process failure follows the fail-stop scheme.[12] Effects of P_j's failure are very similar to those of every P_j's channel failure. Moreover, if P_j owns the token at the time of the failure the token is lost. Some protocol has to discover this event and provide a new token: election algorithms are usually used to perform this task.[2,6] Detection of token loss is not a trivial task, because token loss cannot easily be distinguished from system connectivity loss. If a failure breaks the system in two parts, say A and B for instance, the token being in A, then B discovers that none of its processes owns the token but must not decide that the token was lost and regenerate it. This problem is usually solved by a majority consensus algorithm:[3,15] a process starts executing the election algorithm if there is a majority of processes which agree with it.

Recovery of a process P_i is performed by updating its internal database (*req_array*$_i$, etc.) upon recovery of its outgoing channels: as indicated above, every neighbour of P_i sends its known pending request table.

7.2 Dynamic network reconfiguration

Initial configuration of the system is simple because there is no pending request: every node has to learn the

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018

identity of its incoming channels, but need not learn the number of processes in the system. Reconfiguration of the system while some requests are pending is more difficult. Suppression of a channel has effects similar to channel failure, but some shutdown phase is required in order to empty the channel. Insertion of a channel can be regarded as channel recovery. Suppression of a node must wait until the token leaves the node, or must force it to leave. Node insertion is handled in a way analogous to node recovery.

8. CONCLUSION

The exposed algorithm owns noteworthy features; as we pointed out, it generalises existing algorithms which work on particular topologies (such as ring and complete network) while being as efficient as these algorithms are. Its most interesting and characteristic feature is the absence of particular assumptions on the network topology (apart from the network connectivity requirement) and the fact that no process needs to learn this topology. The algorithm is said to be fully distributed in the following sense: it uses distributed communication techniques (communication by messages); distributed control (there is no central controller); local knowledge only (at any time, no process knows global information such as the network topology).

Acknowledgements

The authors thank an anonymous referee for his valuable comments on the analysis of the algorithm's fault-tolerance.

REFERENCES

1. O. Carvalho and G. Roucairol, On mutual exclusion in computer networks. *Comm. ACM* **26** (2), 147–148 (1983).
2. H. Garcia Molina, Elections in a distributed computing system. *IEEE Trans. on Computers* **C 31**, 48–59 (1981).
3. H. Garcia Molina and D. Barbara, How to assign votes in a distributed system. *J. ACM*, **32** (4), 841–860 (1985).
4. J. M. Helary, A. Maddi and M. Raynal, *Controlling knowledge transfers in distributed algorithms: application to deadlock detection.* Research report INRIA 427 (1985), submitted for publication.
5. L. Lamport, Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, **21** (7), 558–565 (1978).
6. G. Le Lann, *Distributed Systems: Towards a Formal Approach.* IFIP Congress, Toronto (August 1977), pp. 150–160.
7. A. J. Martin, Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming* **5**, 265–276 (1985).
8. M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press (and also North Oxford Academic), 160 pp. (1985).
9. G. Ricart and A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks. *Comm ACM* **24** (1), 9–17 (1981). Corrigendum, *Comm. ACM* **24** (9).
10 G. Ricart and A. K. Agrawala, Author's response to 'On mutual exclusion in computer networks' by Carvalho and Roucairol. *Comm. ACM* **26** (2), 147–148 (1983).
11. R. D. Schlichting and F. B. Schneider, Fail-stop processors: an approach to designing a fault-tolerant computer system. *ACM TOCS* **1** (3), 222–238 (1983).
12. F. B. Schneider, *Paradigms for Distributed Computing*, 468–480. *LCNS* 190. *Springer Verlag, Heidelberg* (1985).
13. W. Stenning, A data transfer protocol. *Computer Networks* **1**, 99–110 (1976).
14. A. S. Tanenbaum, *Computer Networks.* Prentice-Hall, Englewood Cliffs, NJ, 518 pp. (1981).
15. R. W. Thomas, A majority consensus approach to concurrency control for multiple copy databases. *ACM TODS* **4** (2), 180–209 (1979).

Downloaded from https://academic.oup.com/comjnl/article-abstract/31/4/289/380471 by Indian Institute of Technology Hyderabad user on 28 October 2018