A distributed algorithm for mutual exclusion in an arbitrary network

Jean-Michel Hélary, Noël Plouzeau, Michel Raynal

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 496

A DISTRIBUTED ALGORITHM FOR MUTUAL EXCLUSION IN AN ARBITRARY NETWORK

Jean-Michel HELARY
Noël PLOUZEAU
Michel RAYNAL

Mars 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network

Publication 281 - 16 pages - Janvier 86

J.M. Helary, N.Plouzeau, M. Raynal

IRISA
Université de Rennes 1
Campus de Beaulieu
35042 RENNES CEDEX
FRANCE

ABSTRACT – 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.

Index Terms – Distributed algorithms, distributed systems, networks, mutual exclusion, routing protocol, synchronisation.

RESUME - Un algorithme distribué offrant un service d'exclusion mutuelle dans un réseau est présenté. Aucune hypothèse particulière n'est faite sur le graphe qui modélise le réseau si ce n'est sa connexité ; celui-ci peut donc être quelconque. L'algorithme est distribué non seulement par les outils qu'il utilise (communication de messages) et l'absence d'un processus qui jouerait le rôle d'un contrôleur central mais surtout par le fait qu'un site donné n'a jamais besoin de connaître la structure globale du réseau. Il est donc en ce sens plus général que les algorithmes d'exclusion mutuelle distribués qui supposent d'une part un maillage complet ou un anneau et d'autre part la connaissance a priori de cette structure par les divers sites. L'algorithme est prouvé correct et sa complexité analysée (en terme du nombre de messages nécessaires à la réalisation d'une exclusion).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

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. Since 1965 numerous algorithms have been designed either for centralized 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 mean of message transmission). Lamport proposed in his paper on time and logical clocks, as an use of these devices, an algorithm for mutual exclusion in a distributed context [3]. 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 [7] 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 [8] (a synthetic survey of existing mutual exclusion algorithms can be found in [6]).

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.

Section 2 exposes the required system properties, section 3 presents the main principles of the algorithm; the algorithm itself is given in section 4. Section 5 proves some fundamental properties and section 6 evaluates the efficiency of the algorithm.

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 bidirectionnal and is endowed with the following behavioural properties :

- no loss of messages,
- no message alteration,
- finite transmission delay.

Messages don't need to be delivered in the order they were sent. No assumption is made on the network topology, except connectivity.

2.2 Local process knowledge

The only knowledge owned by a process is local, namely 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 transfert control technique has been added, in order to reduce the number of messages required by the message routing protocol **[2]**.

i) Every request sent by a process is propagated in the network with a *flooding broadcast* technique (also called *wave* technique **[9]**). Every time a process receives a request from one of its neighbours it propagates it to its other neighbours, thus broadcasting the request message.

i') The knowledge transfert 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.

ii) 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 **[4]** and **[8]** where the token travels along a ring and a complete network respectively).

iii) 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.

iv) 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 **[3]**, 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 paragraph 2.2).

v) 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 [5]).

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 i') above). The request is added to *P*'s set of known pending requests.
- If the process *P* owns the token and is outside the critical section (see point iii) above), it extracts the oldest request *R* from its known requests set (see point iv) above). It then sends the token to the creator of *R* through *N*, the neighbour of *P* wich sent him the request *R* (see point v) 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 v)).
- A process can enter the critical section only if it owns the token.

4 The algorithm

4.1 Messages

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

a) 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 (point i),i')). The parameters have the following meaning :

- *req_id* unambiguously identifies the request in 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 wich 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 i^{th} subcomponent stores the value that P_i's logical clock had when P_i gave the privilege to another process (see point iii) 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. Every subcomponent of *lud* is initialized with the value -1.

γ) At initialization, 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* 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 just mentionned. These local variables are:

const

- *neighbours*$_i$: **set of** process_id; initialized with the names of P_i's neighbours.

var

- C_i : 0..+∞ **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 initialization time *token_here* if 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_i 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];
    req_array_i[via]:=req_array_i[via]-(elec,x);
    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 referenced (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

Proof

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

$$\text{(ME1)} \qquad Card(\{P_i \text{ such that } in\_CS_i = true\}) \leq 1$$

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

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

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

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

(ME2) and **(ME3)** will prove **(ME1)** and hence theorem one.

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 one.

5.2 Request routing

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 $rt'_i > rt_i$, wich has overtaken R while R and R' were moving in the network ; hence R has been satisfied.

Proof

To prove lemma one 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 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 hands a request over 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 Routing

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 relays the token 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 lemma 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.

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 $lud[i] = -1 \wedge rt_i \geq 0$. If P_i owned the token at least once, it must have leave it in order to send a new request. The statements at line (A) in procedure *transmit_token* implies that $rt_i \geq lud[i] + 1$. 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 $lud[j] \geq rt_j$ (see procedure *transmit_token*). In particular, $lud[orig] \geq rt$; 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 forever 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 have 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 $\gamma_{R'}$ followed by the token and the token transfer is done within a finite delay (lemma 2 and corollary). Thus the request R will become the oldest within a finite delay and process P_i 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 ($1 \leq d \leq n-1$). 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 [5] (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 then we can simplify the algorithm and obtain a new one whose behaviour is similar to the one of Ricart and Agrawala's algorithm [8]. 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 two neighbours. The total request message number lies thus between $n-1$ and $2e$, where e is the number of edges ($e \leq n(n-1)/2$) ; let us point out that this theoretical upper bound is often overevaluated, since the control knowledge transfer technique reduces the actual number of messages in a wave-broadcasting protocol accordingly to the density of the graph [2]. 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 Conclusion

The exposed algorithm owns noteworthy features : as we pointed out, it generalizes 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).

Thanks to these properties of full distribution, the algorithm allows the network to be reconfigured. Only the *neighbours* set of every process has to be updated.

References

[1] Carvalho O., Roucairol G., "On Mutual Exclusion in Computer Networks", Comm. ACM, Vol. 26,2 (February 1983), pp. 147 – 148.

[2] Helary J.M., Maddi A., Raynal M., "Controlling Knowledge Transfers in Distributed Algorithms : Application to Deadlock Detection", Research report INRIA #427, 1985, submitted to publication.

[3] Lamport L., "Time, clocks and the ordering of events in a Distributed System", Comm. ACM, Vol. 21,7 (July 1978), pp. 558 – 565.

[4] Le Lann G., "Distributed Systems: Towards a Formal Approach", IFIP Congress, Toronto, (August 1977), pp. 150 – 160.

[5] Martin A.J., "Distributed Mutual Exclusion on a Ring of Processes", Science of Computer Programming, Vol. 5, (1985), pp. 265 – 276.

[6] Raynal M., "Algorithms for Mutual Exclusion", MIT Press, (1985), 160 p.

[7] Ricart G., Agrawala A.K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Comm. ACM, Vol. 24,1 (January 1981), pp. 9 – 17. Corrigendum, Comm. ACM, Vol. 24,9.

[8] Ricart G., Agrawala A.K., Author's response to "On Mutual Exclusion in Computer Networks" by Carvalho and Roucairol, Comm. ACM, Vol. 26,2 (February 1983), pp. 147 – 148.

[9] Schneider F.B., "Paradigms for Distributed Computing", LCNS #190, Springer Verlag, (1985), pp. 468 – 480.

PI 274 **Détection de pannes et reconfiguration automatique**
Michèle Basseville – 26 pages ; Novembre 85.

PI 275 **Estimation de l'ordre d'un processus Arma à l'aide de résultats de perturbations de matrices**
Jean – Jacques Fuchs – 40 pages ; Décembre 85.

PI 276 **Detection and diagnosis of changes in the eigenstructure of nonstationary multivariable systems**
Michèle Basseville, Albert Benveniste, Georges Moustakides, Anne Rougée – 44 pages ; Décembre 85.

PI 277 **Optimum robust detection of changes in the Ar Part of a multivariable Arma process**
Anne Rougée, Michèle Basseville, Albert Benveniste, Georges Moustakides – 48 pages ; Décembre 85.

PI 278 **Controlling knowledge transfers in distributed algorithms – Application to deadlock detection**
Jean – Michel Hélary, Aomar Maddi, Michel Raynal – 32 pages ; Janvier 86.

PI 279 **Une méthode de conception de programmes fonctionnels**
Raymond Durand, Martine Vergne – 16 pages ; Janvier 86.

PI 280 **Commande de systèmes redondants et évitement d'obstacles**
Bernard Espiau – 52 pages ; Janvier 86.

PI 281 **A distributed algorithm for mutual exclusion in an arbitrary network**
Jean – Michel Hélary, Noël Plouzeau, Michel Raynal – 16 pages ; Janvier 86.

Jean – Michel HÉLARY

Noël PLOUZEAU

Michel RAYNAL

A DISTRIBUTED ALGORITHM FOR MUTUAL EXCLUSION IN AN ARBITRARY NETWORK

Publication interne
n° 281

Janvier 1986