Depth-first traversal and virtual ring construction in distributed systems

Jean-Michel Hélary, Michel Raynal

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 704

DEPTH-FIRST TRAVERSAL AND VIRTUAL RING CONSTRUCTION IN DISTRIBUTED SYSTEMS

Jean-Michel HELARY
Michel RAYNAL

JUILLET 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

DEPTH-FIRST TRAVERSAL AND VIRTUAL RING CONSTRUCTION IN DISTRIBUTED SYSTEMS

Publication Interne n°369 - Juin 1987 - 16 Pages

Jean-Michel HELARY, Michel RAYNAL

I.R.I.S.A.
University of RENNES I
Campus de Beaulieu 35042 RENNES CEDEX

PARCOURS EN PROFONDEUR ET CONSTRUCTION D'ANNEAUX LOGIQUES DANS LES SYSTEMES DISTRIBUES

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)

ABSTRACT. Many distributed systems and applications use decentralized control algorithms lying on a predefinite virtual ring. Such a structure is usually statically defined, thus inherent to the system definition.

Ring construction distributed algorithms are proposed. First, distributed depth-first traversal algorithms are shown to be an algorithmic basis for the construction of such a structure. Then, a ring construction algorithm is presented; its time and message complexities are equal to $2(n-1)$ with $O(n)$ size of messages, where n is the number of proceses. A ring traversal requires $2(n-1)$ messages. This algorithm is then improved : it allows to build a virtual ring whose traversal requires p messages, where $n \leqq p \leqq 2(n-1)$.

RESUME. De nombreux systèmes et applications répartis utilisent des algorithmes de contrôle décentralisé qui s'appuient sur l'existence préalable d'un anneau virtuel; une telle structure est généralement définie statiquement et la définition du système est alors liée à celle d'un anneau particulier.

Nous proposons ici des algorithmes distribués de construction d'anneaux. On montre tout d'abord comment les algorithmes distribués de parcours en profondeur de réseaux (par le parcours séquentiel qu'ils en réalisent) peuvent constituer les fondements algorithmiques pour la construction de telles structures. On présente ensuite un algorithme de construction d'anneau dont les complexités en temps et en messages sont égales a $2(n-1)$, les messages étant de taille $O(n)$, où n est le nombre de sites. Un parcours des n liens de l'anneau virtuel construit requiert $2(n-1)$ messages. L'amélioration proposée ensuite permet de construire un anneau virtuel dont le parcours requiert p messages, avec $n \leqq p \leqq 2(n-1)$.

1 Introduction

A great number of distributed control algorithms (mutual exclusion, election, detection of stable properties, etc.) are based upon a virtual ring structure connecting all the processes of the distributed application. Such a structure provides a very simple way to implement a sequential traversal of processes : a special message, usually known as the *token*, when sent by a process will come back to it after having sequentially visited once and only once every other process. Including all processes in the ring insures this traversal to be fair provided the token keeps the same sense around the ring; recall that fairness is essential to most of distributed control algorithms (mutual exclusion [LL 77,M 85],termination [DFG 83, M 83, CM 86]).

Throughout this paper, we consider a distributed system consisting of n processes connected through bidirectionnal channels allowing direct communication between pairs of neighbour processes linked by a channel. We assume channels are safe, that is messages are delivered without loss or alteration after a finite delay, but they don't need to follow a first-in-first-out rule. The system can be modeled by an undirected graph $G = (X, U)$ whith X the set of processes and U the set of channels; this graph is arbitrary, except connectivity.

Usually rings are built up statically : the definition of each process P_i includes its successor $succ_i$ to which it will send the token after having used it. Should the graph have an hamiltonian cycle, the definition of a ring follows and routing the token from P_i towards $succ_i$ is obvious. If such a cycle cannot be found, the routing function becomes more complex : each process P_i must know :

- on the one hand, when it is visited by the token, through which of its neighbours will the token reach $succ_i$ (the next process to be visited by the token)
- on the other hand, when the token passes P_i on his way to another process, towards which neighbour P_i must transmit it (P_i only takes part to the token routing from a P_j to its $succ_j$).

Definition of the routing function appears to be the heart of the problem.

We are interested in the design of distributed algorithms allowing dynamic built-up of ring and associated routing in an arbitrary network. We will assume that processes have distinct identities, but each process knows only its own identity together with those of its neighbours : no global network information, such that the number n of processes or the structure of G, is known by the processes.

The paper has three parts : §2 deals with the monitoring of a ring; in §3, a distributed algorithm building up a ring is proposed; moreover, distributed depth-first traversal techniques are shown to be a good basis to the derivation of ring building-up algorithms; the derived algorithm is improved in §4.

2 Ring traversal

A virtual ring can be defined as a circular list on the set X of processes; each process P_i is thus endowed with the variable $succ_i$ which defines its logical successor on the ring. When the token is located on a process, as announced in the introduction, two cases have to be distinguished : the token visits the process, or the token is only going through the process on its way to the next process to be visited. To this end, the token carries on the identity of its addressee, thus having the following form : *token(addr)*.

When receiving *token(addr)* from its neighbour P_j, P_i scans the field *addr* and, if $addr = i$ learns that it is visited; having processed, P_i sets the *addr* field to $succ_i$ before sending the token away. If on the contrary $addr \neq i$, P_i immediatly transmits the token without using it or modifying the field *addr*. The neighbour P_k towards P_i sends the token can be locally maintained through a routing table R_i such that $R_i(j) = k$; this table locally implements the token routing function. This can be summarized by the algorithm of figure 1.

```
                Text of process P_i
Upon reception of token(addr) from neighbour P_j
  do
    if addr = i then < use the token>
                     addr := succ_i
    endif ;
    k := R_i(j) ;
    send token(addr) to neighbour P_k
  enddo
```

Figure 1.

The task of a ring build-up algorithm consists in the definition of variables $succ_i$ and R_i on each process of the system.

Remark : other ring monitoring techniques are possible; for instance,the token could be free from carrying control information such as *addr*; yet each process P_i should then have a local variable $token\_from_i$ indicating from which neighbour the token will be received when its addressee is P_i (if the token is received from another neighbour, P_i is only a step on the way to the token addressee).

3 Building up a ring

Dynamic ring construction could be achieved rather simply using the following method : a particular process P_0 collects from other processes information relative to the network topology, then uses a centralized graph algorithm to compute a ring and the corresponding routing; finally, P_0 sends down ring informations relative to each process. Although initial data and results are distributed over the network, computation is centralized in the particuliar process P_0. (A network spanning tree rooted in P_0 can be used for the purpose of data collection and results diffusion).

We are here interested in a distributed solution. A ring being characterized by a total ordering of the set of processes, such solutions can be based on distributed sequential traversal algorithms, and particularly distributed *depth-first* traversal algorithms, initiated by a process P_k [AWE 85, CHE 83, SEG 83]. Principles of such traversal algorithms are at first stated, then applied to the ring construction problem.

3.1 Depth-first traversal

3.1.1 Principle

Distributed depth-first algorithms own the following property : control flow of the traversal is unique (alike coroutines) and located at a given time in a process or in transit on a channel (taking the form of a message). The process owning the control is said to be *active* — otherwise *passive* . Initially, the active process is P_k. When a process P_i becomes *active for the first time*, it becomes *visited* and defines the process from which the control was obtained as its father, we say that P_i *joins the traversal*. Moreover, when P_i is active, it shifts the control to a non visited neighbour if any, through a DISCOVER message, whilst if all its neighbours are visited, P_i moves back control to its father through a RETURN message or, if $i = k$, terminates the traversal algorithm. Thus, at least $2(n-1)$ control shifts are necessary, a lower bound for time complexity [LMT 87].

Implementation of this traversal technique requires, for the active process to know exactly which of its neighbours are visited. Below we present two algorithms satisfying this constraint.

3.1.2 Awerbuch's algorithm [AWE 85]

Under the constraint that all messages are of fixed length independant of the network, Awerbuch has presented an algorithm which requires, apart from messages DISCOVER and RETURN, two other kinds of messages VISITED and ACK allowing each visited process to know which of its neighbours are visited and to know that all its neigbours learnt indeed that it is visited.

When a process P_i joins the traversal, it sends to each of its neighbours— but its father— a message VISITED, answered back by messages ACK. When P_i has received all expected ACK,

it knows that all its neighbours learnt that it is now visited and can thus proceed to shift the control (by sending DISCOVER or RETURN).

Time complexity of Awerbuch's algorithm is $4n - 2$ and message complexity is $4m$, m being the number of channels. This algorithm has recently been improved, by eliminating the ACK messages [LMT 87].

3.1.3 Another algorithm

Awerbuch's algorithm insures knowledge of visited neighbours by synchronization, controlling progress of messages DISCOVER and RETURN, with the help of messages VISITED and ACK. This synchronisation can be removed, and the message complexity be consequently reduced down to $O(n)$ without increasing time complexity; only messages DISCOVER and RETURN remain. The price to be paid is to put into these messages a control field allowing an active process to know its visited neighbours : it consists of the set Z of visited processes. (Carrying over such a global control information is necessary since any process owns initially only a local information relative to the network). The size of messages, although dependent of n, remains bounded in $O(n)$ [HPMR 87,HMR 87]. A process P_i is then able, at the time when it becomes active, to determine the set of its non ¡visited neighbours, equal to $neighbours_i - Z$ ($neighbours_i$ denotes the set of neighbours of P_i).

The text of this algorithm for a process P_i is given in figure 2. P_i is endowed with the variable $father_i$ denoting the identity of the process having sent a DISCOVER message to P_i. $father_k$ has the value k for the initiator P_k. The latter initiates the algorithm by self-transmission of the message DISCOVER(∅).

It is easy to see that time and message complexities are both equal to $2(n - 1)$; moreover the length of messages is bounded up by n process identities.

Remark. A single message type should be sufficient to implement this algorithm, acting now like DISCOVER now like RETURN. To this end each process P_i must be endowed with a boolean $visited_i$ initialized to **false**. When P_i joins the traversal, it sets $visited_i$ to **true** and performs the statement corresponding to DISCOVER; upon subsequent receptions, the statement corresponding to RETURN is performed.

3.2 Application to ring construction

3.2.1 Principle

Recall that at the end of the traversal each process has to know its logical successor $succ_i$ in the ring and its local routing table R_i.

The depth-first traversal (according to the principle exhibited in §3.1.1) can be seen as the moving of a "scanning token" (the control) from a process to another, issued from and coming

```
upon reception of DISCOVER(Z) from neighbour P_j
  do (* P_i joins the traversal *)
    father_i := j ;
    let Y = neighbours_i − Z ;
    case
      Y ≠ ∅ → select x in Y ;
              send DISCOVER(Z ∪ { i }) to neighbour P_x

      Y = ∅ → send RETURN(Z ∪ { i }) to neighbour P_j
    endcase
  enddo

upon reception of RETURN(Z) from neighbour P_j
  do
    let Y = neighbours_i − Z ;
    case
      Y ≠ ∅ → select x in Y ;
              send DISCOVER(Z) to neighbour P_x

      Y = ∅ and father_i ≠ i
            → send RETURN(Z) to neighbourP_father_i

      Y = ∅ and father_i = i
            → P_i is the initiator and the algorithm is terminated
    endcase
  enddo
```

Figure 2.

back to the initiator P_k. This scanning token visits all processes according to a preorder traversal of the network spanning tree rooted at P_k built by the algorithm (this tree is materialized by the variables $father_i$). The order in which processes join the traversal is total (the traversal is sequential), thus defining a virtual ring on the network.

The location of each process in this virtual ring is obtained through the knowledge of the process situated either after it in the virtual ring (logical right neighbour) or before it in the virtual ring (logical left neighbour). Now the logical left neighbour can be easily obtained by a P_i by letting the scanning token (i.e. DISCOVER and RETURN messages) to carry the identity of the last process having joined the traversal. Managing this control information is straightforward :

- Upon receiving a DISCOVER message, the active process P_i stores the process identity carried by this message as its logical left neighbour, then lets the scanning token sent out (under DISCOVER or RETURN form) carry P_i's identity.
- Upon receiving a RETURN message P_i either passes on the scanning token without altering the carried identity or, if $i = k$ and detects the traversal termination it stores the received identity as its logical left neighbour (thus closing the ring).

Such a virtual ring can be operated by moving a token from a process P_i to its successor $succ_i$ defined as its logical left neighbour (stored by P_i).

Finally, the routing problem can be solved : a complete tour around the virtual ring is implemented exactly by the opposite of the scanning token traversal. Consequently when it receives the scanning token from P_j and sends it to P_k, P_i sets up $R_i(k)$ to j.

As can be seen this ring building method lies in enriching a depth-first traversal algorithm with statements generating values for the $succ_i$ and R_i variables. We apply it to the second traversal algorithm presented (§3.1.3).

3.2.2 A ring construction algorithm

Context of a process P_i

Besides the variable $father_i$ required by the traversal, each process P_i is endowed with the following variables defining its participation to the virtual ring : $succ_i$ and R_i as defined previously (recall that $R_i(j) = k$ iff the token received from P_j has to be sent to P_k). Moreover, the initiator P_k must have a variable pv to store the identity of the first P_k's neighbour to which a DISCOVER message is sent, allowing to set up the last routing link (when the traversal terminates).

The text of the algorithm consists of the program performed by any process P_i; it is given in figure 3.

To start up the algorithm,the initiator P_k has to perform the following prelude :

```
father_k := k ;
select x in neighbours_i ;
pv := x ;
send DISCOVER({k},k) to neighbour P_x
```

Obviously this construction algorithm inherits time and message complexities from the underlying depth-first traversal algorithm. Moreover, moving a token along the n logical links of the virtual ring requires $2(n-1)$ messages.

```
text for a process P_i

upon reception of DISCOVER(Z,last) from neighbour P_j
  do
    succ_i := last ; father_i := j ;
    let Y = neighbours_i - Z ;
    case
      Y ≠ ∅ → select x in Y ;
              R_i(x) := j ;
              send DISCOVER(Z ∪ { i } , i) to neighbour P_x

      Y = ∅ → R_i(j) := j ;
              send RETURN(Z ∪ { i } , i) to neighbour P_j
    endcase
  enddo

upon reception of RETURN(Z, last) from neighbour P_j
  do
    let Y = neighbours_i - Z ;
    case
      Y ≠ ∅ → select x in Y ;
              R_i(x) := j ;
              send DISCOVER(Z,last) to neighbour P_x

      Y = ∅ and father_i ≠ i  (* P_i is not the initiator *)
          → R_i(father_i) := j ;
            send RETURN(Z, last) to neighbour P_father_i

      Y = ∅ and father_i = i (* P_i is the initiator :
                                construction terminates *)
          → succ_i := last ;
            R_i(pv) := j
    endcase
  enddo
```

Figure 3.

4 An improvement of the ring construction algorithm

4.1 Principle

Some networks display virtual rings requiring less than $2(n-1)$ messages to move the token round their n virtual links. This happens for example in the case of networks including an hamiltonian cycle implementing an optimal ring whose traversal requires exactly n messages. (Let us recall that to determine an hamiltonian cycle in an arbitrary graph is a NP-complete problem).

In this section we are interested in improving routing towards a message complexity at most equal to $2(n-1)$. Moreover, as we will see, such an improvement will be obtained by an algorithm whose time and message complexities are at most $2(n-1)$. Improvement lies in using some network channels to shorten the return path during the traversal (such channels are used as "ring chords").

Figure 4 shows an example : the traversal issued from P_1 has visited successively P_2, P_3, P_4 (through DISCOVER messages); P_4 may then send a RETURN message directly over the channel (P_4, P_2) which constitutes a short-cut (chord) of the return path towards P_1 (from which the traversal will go further on).

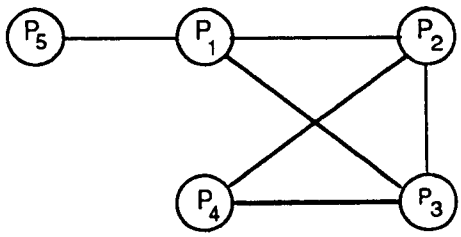

Figure 4. Example

4.2 Implementation

In order to implement this principle, the active process in position to send back a RETURN message has to know :

- The return path up to the process from which traversal will be able to go further on (or terminate),

- Existing chords shortening this path.

To this end control messages will carry the following informations :

- The set Z containing the identities of the visited processes, now organized as a sequence,
- Associated to each element of Z, the list of its non visited neighbours.

Such an information structure will be denoted *LZ*. (Note that a process is usually unable to learn the whole network structure from this information alone).

The active process can update this information structure with the following primitives :

insert $(i, listofprocesses)$: includes the identity of P_i and a list of its neighbours at the end of *LZ*.

remove (x) : removes from *LZ* the process identity x everywhere it appears as a non visited neighbour.

The active process can read information from *LZ* with the two following functions :

last_pending = if exists within *LZ* at least one process identity endowed with a non empty list of non visited neighbours
then (α, β) where α is the last such identity in the sequence and β is the first element in the list of the nonvisited neighbours of P_α
(Note that we have $\alpha \neq \beta$)
else nil
endif

The function *last_pending* is used by a process P_i joining the traversal when the latter cannot go further on from P_i; P_β will then be the predecessor of P_i on the ring (right and left ring links are thus available on each process).

first_neighbour_to (α : **identity_of_process**) = α itself if P_α is a neighbour of P_i, or the identity s of the first P_i's neighbour appearing after α in *LZ*.

This function allows P_i to determine a short-cut towards P_α from which the traversal can go on.

Control messages DISCOVER and RETURN carry now the following information :

- *LZ* (instead of the set *Z* in the algorithm 3.3.2),
- *last* (as in the algorithm 3.3.2).

Moreover, RETURN messages carry the couple of process identities (α, β) returned by the function *last_pending* (allowing computation of short-cuts when the traversal goes back to P_α). The text of the algorithm is given in figures 5.a and b.

The algorithm is initiated by P_k by the following statement :

let l = the list of P_k's neighbours except one of them, P_x ;
pv := x ;
send DISCOVER ((k, l), k) **to neighbour** P_x

Remark. The variable *father*$_i$ used in the first-depth traversal (§3.1.3) and in its enriched version which builds up a ring (§3.2.2) is no more needed here. Indeed RETURN mesages do not necessarily travel along the branches of a tree; they can take short-cuts thanks the information conveyed by *LZ*.

4.3 Example.

Consider the network of figure 6 : Values of the variables $succ_i$, $pred_i$ and R_i computed by the distributed algorithm for each P_i are displayed below in table 1. (It is supposed that in the corresponding computation, for a given process the list of its neighbour's identities is an increasing one; moreover P_1 is the initiator of the traversal).

Table 1.

Processes	$succ_i$	$pred_i$	R_i
1	12	2	2 → 6 6 → 2
2	1	3	3 → 1 1 → 4
3	2	4	4 → 2
4	3	6	2 → 3
5	6	8	8 → 6
6	4	5	5 → 1 1 → 8
7	9	10	9 → 9
8	5	9	9 → 5 6 → 9
9	8	7	7 → 8 10 → 7 8 → 10
10	7	11	11 → 9 9 → 12
11	10	12	12 → 10
12	11	1	10 → 11

As shown by the routing table distributed over the processes, there are three logical links of the

```
upon reception of DISCOVER(LZ,last) from neighbour P_j
  do
    succ_i := last ;
    let Y = set of non visited P_i's neighbours ; (* known from LZ *)
    case
     Y ≠ ∅ →
         let x = head of list l obtained by ordering Y ;
         remove x from l ;
         pred_i := x ; (* P_x is the next process to join the traversal *)
         R_i(x) := j ;
         insert (i, l) ;  (* P_i indicates that it joins the traversal
                             and includes its non visited neighbours *)
         remove(x) ;
         send DISCOVER(LZ,i) to neighbour P_x

     Y = ∅ →
         case last_pending = (∝, β) →
              pred_i := β ; (*P_β is the next process to join the traversal *)
              s := first_neighbour_to(∝) ;
              R_i(s) := j ;
              insert(i, empty list) ;  (* P_i indicates that it joins the traversal *)
              send RETURN(LZ,i,∝, β) to neighbour P_s

            last_pending = nil →  (* P_i detects the end of traversal *)
              ∝ := head of LZ ; (* P_∝ is the initiator *)
              pred_i := ∝ ;
              s := first_neighbour_to(∝) ;
              R_i(s) := j ;
              remove from LZ all elements from s to the end ;
              send RETURN (LZ,i,∝, ∝) to neighbour P_s
         endcase
    endcase
  enddo
```

Figure 5. a

virtual ring for which the corresponding implementation requires more than one network channel :

```
upon reception of RETURN(LZ,last ∝, β) from neighbour P_j
  do
    case
      ∝≠ i →
          s := first_neighbour_to(∝) ; (* (P_i, P_s) is a chord on the
                                          path P_i to P_∝ *)
          R_i(s) := j ;
          send RETURN(LZ, last, ∝, β) to neighbour P_s

      ∝ = i →
          case
            ∝≠ β→ (* P_i has non visited neighbours : P_β is the first one *)
                  R_i(β) := j ;
                  remove(β) ;
                  send DISCOVER(LZ,last) to neighbour P_β

            ∝ = β→ (* P_i is the initiator and the ring construction is terminated *)
                  succ_i := last ;
                  R_i(pv) := j
          endcase
    endcase
  enddo
```

Figure 5. b

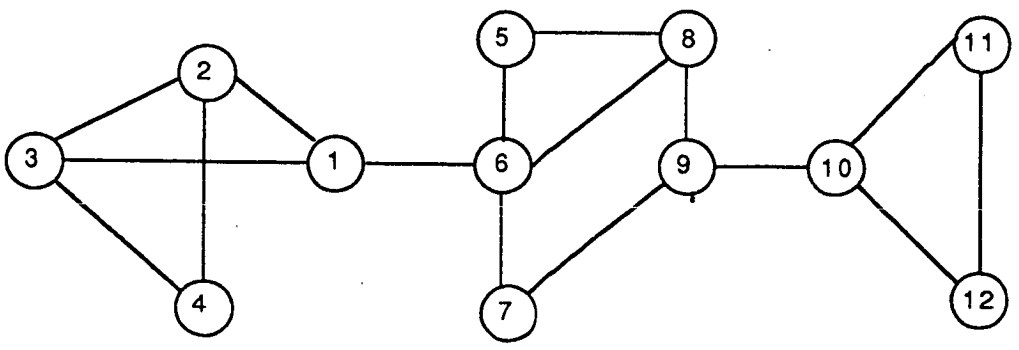

Figure 6.

6,4	implemented by the path	6,1,2,4
10,7		10,9,7
1,12		1,6,8,9,10,12

5 Conclusion

This paper has displayed the interest of networks depth-first traversal in order to build up virtual rings. The benefice of such structures is no longer to be proved with regard to control algorithms encountered in distributed systems and applications.

The traversal algorithm presented has time and message complexities of $2(n-1)$, the size of each message being $O(n)$. When process identities are integers and their bound is known by each process, messages can be easily implemented by an array of bits [SEG 83].

We have seen how statements generating value for local variables can be grafted on such traversals to build a virtual ring. $2(n-1)$ messages are required for the token to move around this ring. The proposed improvement allows to reduce this number by defining short-cuts for the token moves; the price to be paid is then longer messages during the construction of the virtual ring.

References

[AWE 85] AWERBUCH,B., *A new distributed depth-first-search algorithm,* Information Processing Letters, Vol. 20,3 (1985), pp. 147-150.

[CM 86] CHANDY,K.M., and MISRA,J., *An exemple of stepwise refinement of distributed programs : Quiescience detection,*ACM Toplas, Vol.8,3, (july 1986) pp326-343.

[CHE 83] CHEUNG,T., *Graph traversal techniques and the maximum flow problem in distributed computation,* IEEE Trans. on Soft. Eng., Vol. SE-9, n° 4, (july 1983), pp. 504-512.

[DFG 83] DIJKSTRA,E.W. , FEIJEN,W.H.J. and VAN GASTEREN,A.J.M., *Derivation of a termination algorithm for distributed computations,* Information Processing Letters, Vol.16, (1983), pp.217-219.

[HPMR 87] HELARY,J.M., MADDI,A., PLOUZEAU,N. and RAYNAL,M., *Parcours et apprentissage dans un reseau de processus communicants,* TSI, Vol.6,2, (1987), pp127-140.

[HMR 87] HELARY,J.M., MADDI,A. and RAYNAL,M., *Controlling information transfers in distributed algorithms : application to deadlock detection,* Int. Conf. on Parallel Processing and Applications, L'Aquila (Italy), (Sept. 1987).

[LMT 87] LAKSHMANAN,K.B., MEENAKSHI,N. and THULASIRAMAN,K., *A time optimal message-efficient distributed algorithm for depth-first-search,* Information Processing Letters, Vol.25, (1987), pp.103-109.

[LL 77] LE LANN,G., *Distributed systems : towards a formal approach,* IFIP Congress, Toronto, (August 1977), pp 155-160.

[M 85] MARTIN,A.J., *Distributed mutual exclusion on a ring of processes,* Science of Computer Programming, Vol.5, (1985), pp 265-276.

[M 83] MISRA,J., *Detecting termination of distributed computations using markers,* Proc. 2nd ACM Conf. on PODC, (August 1983), pp.290-294.

[SEG 83] SEGALL,A., *Distributed networks protocols,* IEEE Trans. on Inf. Theory, Vol. IT29,2, (1983), pp 23-35.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique