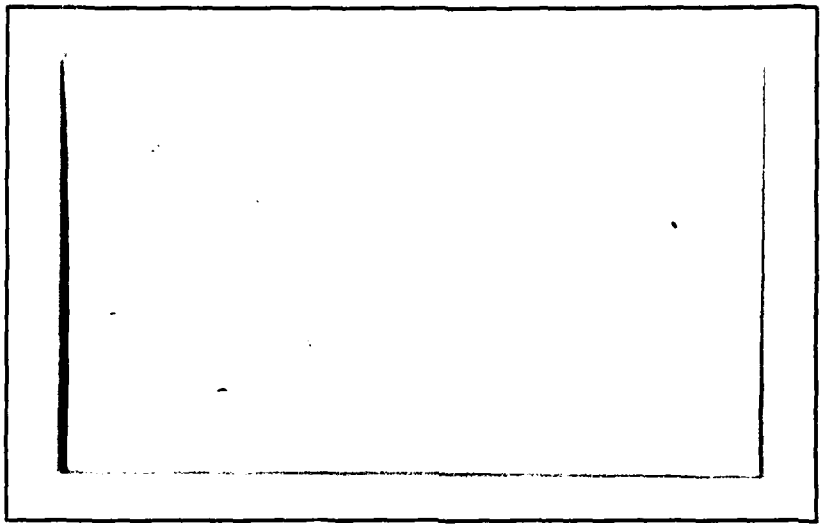
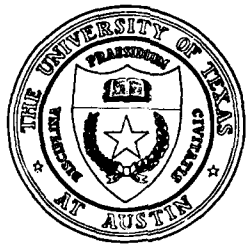
AD-A120 871 TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES F/G 9/2
A DISTRIBUTED DEADLOCK DETECTION ALGORITHM AND ITS CORRECTNESS --ETC(U)
FEB 82 K M CHANDY, J MISRA, L HAAS AFOSR-81-0205
UNCLASSIFIED TR-LCS-8204 AFOSR-TR-82-0882 NL

AFOSR-TR- 82-0882

(3)












THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF COMPUTER SCIENCES

AUSTIN, TEXAS 78712

# A DISTRIBUTED DEADLOCK DETECTION ALGORITHM AND ITS CORRECTNESS PROOF*

K. M. Chandy
J. Misra
L. Haas

TR-LCS-8204 February 1982



Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712



* This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR 81-0205.

# A DISTRIBUTED DEADLOCK DETECTION ALGORITHM AND ITS CORRECTNESS PROOF*

by

K. M. Chandy, J. Misra, L. M. Haas
Computer Sciences Department
University of Texas at Austin
Austin, Texas 78712
(512) 471-4353

Abstract

This paper presents a very simple distributed algorithm for deadlock detection in a network of processes. The algorithm is proven correct, i.e., we show that all true deadlocks are detected and no false reporting of deadlock occurs. In ~~our~~ algorithm no process maintains global information. All messages have identical length and are short, consisting of a single node name and a sequence number. ~~Our~~ work is based on the work of Dijkstra and Scholten on termination detection of diffusing computations.

Submitted May '81
Revised Feb '82



* This work has been partially supported by the Air Force Office of Scientific Research under grant AFOSR 81-0205.

## 1. INTRODUCTION

There has been a great deal of interest in networks of processes where communication is through messages. Hoare's CSP [7] and Ada [15] are examples of models of such networks. Hoare has shown that many operating system features can be implemented using CSP. We assume a simple model of distributed computation; most of the message passing primitives which have been presented in the literature can be implemented in terms of our model. We present a simple distributed algorithm to detect deadlocks in (our model of) message-communicating networks of processes.

The structure of the paper is as follows: Our model of computation and deadlock is given in section 2; the core of the algorithm appears in section 3 and its proof in section 4. Section 5 is a discussion of the complete algorithm. Implementation issues are discussed in section 6.

Most of the previous work in this area has been concerned with deadlocks due to resource allocation [1,3,4,8,9,10,11,13] . In our model, as in CSP, resources must be implemented by processes; requests for resource allocation and release must be implemented by messages. In a resource allocation model a process cannot proceed with execution until it receives all the resources that it is waiting for. In CSP, ADA and similar models, a process cannot proceed with its execution until it can communicate with at least one of the processes it is waiting for. For instance a process in CSP, executing a guarded command may wait to receive from

several processes; a guard succeeds and execution continues only when a message is received from one of these processes. The difference between the resource model and our model is between "waiting for all resources" and "waiting for any one message"; this difference results in very different algorithms for the two models.

Dijkstra and Scholten presented an algorithm to detect termination in diffusing computations [2]. Diffusing computation is a model of distributed computations in which the computation is started by a special process, the initiator, sending one or more messages. Processes other than the initiator can send messages only after receiving the first message. Each process waits to receive messages from all other processes at all times. Thus the computation terminates only when every process is idle waiting for every other process. Our model is intended to support implementations of languages such as CSP and ADA, and therefore we must allow (1) a process to wait selectively for some (not necessarily all) other processes and (2) any process can send a message without having received a message. As a consequence, we must detect deadlock in our model when a subset of processes wait only for each other whereas in the Dijkstra-Scholten model termination is detected only when all processes are waiting for all others. Our algorithm is based upon the work of Dijkstra and Scholten; we are grateful to them for their continuing advice and encouragement.

## 2. A MODEL OF DISTRIBUTED COMPUTATION: A NETWORK OF PROCESSES

A network consists of a set of processes which communicate with one another exclusively by messages. We adopt the message communication protocol of Dijkstra and Scholten [2]: we require that any message sent by one process to another is received correctly after an arbitrary but finite delay and that message transmissions obey the following first-in-first-out rule: messages sent by any process i to any other process j are received by j in the sequence in which they are sent by i. These requirements can be met by using sequence numbers and time-outs [14] and by having every process poll periodically for input messages.

A process i can only assert that any message it sends to process j will be received eventually. However, process i cannot assert that j has actually received the message unless it recieves some form of acknowledgement from j. The operation of sending a message requires no synchronization with the receiving process, unlike Communicating Sequential Processes of Hoare [7]; therefore a process never waits to send a message. The CSP protocol is implemented by having the sender of a message first send a message and then immediately wait for an acknowledgement and by having the receiver of a message immediately send an acknowledgement.

At any time a process is in one of two states, idle or executing. Only executing processes can send messages. An executing process may change its state (to idle) at any time.

Associated with every idle process is a set of processes, called its dependent set. An idle process becomes executing upon receiving a message from any process in its dependent set; it does not change state nor its dependent set otherwise. A process is terminated if it is idle and its associated dependent set is empty. For the moment, we assume that processes do not terminate, fail or abort; there issues are considered in section 6.

## 2.1 System Properties: SP1 to SP4

We summarize system properties:

SP1) A process receives messages from another process in the sequence in which they were sent.

SP2) On receiving a message from a process in its dependent set, an idle process becomes executing, otherwise an idle process remains idle and does not change its dependent set.

SP3) An executing process may become idle without receiving a message.

SP4) Only executing processes can send messages.

## 2.2 Deadlock

Intuitively, a nonempty set S of processes is deadlocked if all processes in S are "permanently" idle. This paper presents an algorithm which detects deadlock based solely on the process states; idle or executing.

It is not possible to detect deadlock in the following situation: process A is waiting for a message from process B; process B will send a message to process A only upon completion of execution of a loop. Process A is deadlocked if and only if process

B's loop computation is nonterminating. Detection of such a deadlock amounts to solving the halting problem and hence is unsolvable. We must perforce assume in this situation that A is not deadlocked since B may send it a message some time in the future. Therefore we adopt the following operational definition of deadlock.

A nonempty set of processes S is deadlocked if and only if,

(1) all processes in S are idle and

(2) the dependent set of every process in S is a subset of S, and

(3) if i is in S and j is in i's dependent set then process i has received every message sent to it by process j.

A process is deadlocked if it belongs to some deadlocked set.

A non-empty set S of processes satisfying the above three conditions must remain idle permanently because (1) an idle process i can become executing only upon receiving a message from some process j in its dependent set (SP2), (2) every process j in i's dependent set is also in S and therefore cannot send a message (while remaining in the idle state - from SP4) and there are no messages in transit from j to i.

## 3. A DISTRIBUTED DEADLOCK DETECTION ALGORITHM

We now describe an algorithm which allows an idle process to determine if it is deadlocked. The process does so by initiating a query computation when it enters the idle state. The query computation is distinct from the underlying computation for which the deadlock is being detected. Processes may exchange

messages for the query computation even in the idle state: the idle state only refers to the underlying computation. The process which initiates the query computation is called the initiator.

Several processes may initiate query computations and the same process may initiate query computations several times. To distinguish query computations from one another, the variables and messages in the n-th computation initiated by process i are tagged with (i,n). In this section we shall describe a single, generic query computation, say the (i,n)-th. For brevity, we shall not write the tag (i,n) explicitly; its existence should be assumed.

A query computation uses two kinds of messages: query and reply. There will be at most one query sent from a process i to a process j in a query computation. After j gets a query from i, j may send at most one reply to i. The initiator starts the computation by sending queries to all processes in its dependent set.

The query computation will have the following properties:

(D1) If the initiator is deadlocked when it starts the query computation then it will receive replies for all the queries that it sends (see theorem 1).

(D2) If the initiator has received replies for all the queries that it sent, then it is deadlocked (see theorem 3).

In our algorithm each process i uses a local variable called engager(i); this is initially undefined and is set to the identity

of the process from which an idle process receives its first query, which we call the engaging query. Once the engager is set to the identity of a process, its value is never changed. The time at which the engager is set is called the time of engagement and the process is said to be engaged from that time onwards. engager(i) is said to engage i.

## 3.1 Algorithm

### 3.1.1 Algorithm for process i (other than initiator)

If process i is executing, discard all queries and replies received. If process i is idle:

(A1) Upon receiving the first query:
set engager to the identity of the process which sent the query {become engaged} and send queries to all processes in i's dependent set.

(A2) Upon receiving a subsequent query:
if the process has been idle continuously (i.e. never become executing) since engagement, then send the reply to this query.

(A3) Upon receiving a reply:
if the process has been idle continuously since engagement, and replies have been received from all processes in its dependent set, then send the reply to the engager.

### 3.1.2 Algorithm for the initiator

It is convenient to imagine that the initiator process initially receives a fictitous query from a fictitous external

process thus causing it to become engaged and initiate the computation. The initiator then follows the above algorithm except that instead of sending the reply back to its fictitous engager, it declares itself "deadlocked".

3.2 Query Computation Properties: QCP 1 to QCP 5

QCP 1) The number of queries and replies in a query computation is bounded.

Proof: At most one query and one reply is sent from any process to any other process in the network in a query computation, and there are only a finite number of processes.

QCP 2) If a process sends a reply to a query then it must have been continuously idle since its engagement. Furthermore, if it replies to its engager then it has received replies to all queries it sent. (From A2,A3)

QCP 3) A process that never becomes executing after the time of its engagement will reply to all nonengaging queries (A2). Furthermore, it will reply to its engaging query if it receives replies to all its queries (A3).

QCP 4) Only idle processes can become engaged and only engaged processes send queries or replies. (From A1)

QCP 5) A process sends queries only to processes in its dependent set. (From A1)

Note that a process does not participate in query computations while it is executing.

## 4. PROOFS OF CORRECTNESS

We now prove that the query computation has the properties (D1) and (D2) presented in section 2.

Lemma 1

If a process i never replies to a query sent to it at time t, then either

(1) i becomes executing after receiving the query, or

(2) there exists a process j in i's dependent set at t, which never replies to a query sent to it at t', t' > t.

Proof: We prove the lemma by showing that if (1) is false, then (2) must be true. If i is sent a query at time t to which it does not reply and it is idle from time t onwards, then the query must be an engaging query, because nonengaging queries are replied to immediately (QCP 3). If i never replies to this engaging query then it can only be because i never receives a reply to a query that it sent to some process j (QCP 3) in its dependent set. i must have sent the query to j only after receiving its own engaging query (QCP 4) and hence after t.

Lemma 2

If a process i becomes engaged and subsequently becomes executing at time t, then either

(1) i never replies to its engager, or

(2) there exists a process j in i's dependent set which becomes engaged and subsequently becomes executing at time t, t' < t.

Proof

Outline: We show that if (1) is false then (2) must be true for some j in i's dependent set. We show that the following sequence

of events must have happened:

at i: i becomes engaged, i sends queries to all processes in its dependent set including j, i receives replies from all processes in its dependent set including j, i replies to its engager, i receives a message m from j; i becomes executing.

at j: j receives a query from i (j was engaged previously or becomes engaged now), j sends a reply to i, j becomes executing, j sends the message m to i.

It follows from this sequence of events that j becomes executing before i does and after j became engaged. We now show that this sequence of events must have occurred.

Suppose process i becomes executing at t after its engagement and i also sends a reply to its engager. Then it must have been continuously idle in the interval since its engagement and up to the time at which it replies to its engager (QCP 2). Therefore i becomes executing only after sending the reply to its engager. In order for i to send a reply to its engager, it must have received replies from all processes in its dependent set (QCP 2). In order for i to become executing it must have received a message from some process j in its dependent set (SP 2).

We now show that process j became executing at t', after its engagement and $t' < t$. Since i became executing after sending the reply to its engager, it must have received the reply from j before it received the message from j. Therefore j must have sent the reply to i before sending it the message (SP 1).

j must be engaged and must have been continuously idle since its engagement, when it sent the reply to i (QCP 2). j must have been executing when it sent the message to i (SP 4). Therefore j must have become executing after it became engaged. Since i became executing at t as a *consequence* of receiving j's message, j must have become executing at some t' < t.

Theorem 1

If the initiator of a query computation is deadlocked when it initiates the computation, it will eventually declare itself "deadlocked," i.e. it will receive replies to all queries it sent.

Proof: Consider a set of processes S, including the initiator which is deadlocked at the initiation of query computation. From the definition of deadlock, the dependent set of every process in S, is included in S. Only processes in S can receive queries (QCP 5). Since all processes in S are deadlocked, no process i in S can ever become executing; therefore, condition (1) of lemma 1 does not apply to any query sent to any process. Hence, if a process i never replies to a query sent to it at time t', then there is some other process j which never replies to a query sent to it at a later time t'. Using this result inductively and the fact (QCP 1) that the number of queries is bounded the theorem follows.

Theorem 2

If the initiator of a query computation declares itself "deadlocked", then it belongs to a deadlocked set.

Proof: Let S be the set of processes including the initiator which received queries during the computation. We will show that S is a deadlocked set.

Every process replying to its engager must have received replies for all queries it sent. From this fact the following inductive hypothesis can be established: if the initiator declares itself deadlocked, a reply must have been received to the i-th query in the computation, for i = 1,2,... Therefore every process in S replies to its engager. Hence condition (1) of Lemma 2 does not hold for processes in S. Therefore if process i in S becomes executing at t after its engagement then some process j in S becomes executing at t' after its engagement and t' < t. Using this inductively, it follows that no process in S can become executing after its engagement and hence S is a deadlocked set.

## 5. THE GENERAL DEADLOCK DETECTION ALGORITHM

Whenever a process becomes idle there is a possibility that it may be deadlocked. Therefore, every process must initiate a new query computation every time it becomes idle. To distinguish the different query computations, the n-th query computation initiated by process i is tagged with the identifier (i,n), i.e. all replies, queries and variables associated with that query computation are tagged with (i,n).

Suppose a process i initiates a query computation (say the n-th one), and subsequently initiates another query computation. To initiate the (n+1)-th query computation, the process must have left the idle state corresponding to the n-th query computation

and must have initiated the (n+1)-th query computation on next entering an idle state; hence the process could not have been deadlocked when it initiated the n-th query computation. Therefore, it is sufficient for every process to keep track of only the latest query computation initiated by any other process j. It may discard any query or reply that it receives from an earlier query computation. Hence every process i needs to keep track of at most N parallel query computations where N is the number of processes in the network. A process may be simultaneously participating in as many as N query computations and may therefore have N engagers, corresponding to each query computation.

Theorem 3

At least one process in every deadlocked set will report "deadlocked" by the general deadlock detection algorithm.

Proof: The last process to become idle in a deadlocked set of processes must initiate a query computation when it becomes idle. From theorem 1, this process will report "deadlocked."

## 6. ASPECTS OF THE ALGORITHM

This algorithm seems attractive for performance as well as correctness reasons, since the overhead of query computations and message traffic associated with query computations is generated primarily when processes are idle (i.e. have nothing to do and nothing to send). Furthermore, executing processes need only discard queries and replies. Every single query computation involves no more than e queries and replies where e is the number of communicating process pairs in the network. To reduce the number of

query computations, a process may initiate a query only if it has been idle continuously for some time T, where T is a performance parameter. If the process leaves the idle state before T, we have avoided initiating a query computation. Time-outs may be used in a similar manner for query and reply propagation. Note that since every process could initiate one or more query computations, proper choice of T may be critical in reducing the number of query computations. Issues related to this can be found in [ 5 ].

Every query computation will cease in finite time because the number of queries and replies associated with a query computation is bounded and the time required to process a query or reply is finite. If a process initiates a query computation when it is deadlocked, the computation will cease only after all queries and replies have been processed. If the process initiates a query computation when it is not deadlocked, the computation will cease, but some queries or replies may be discarded.

Every deadlocked process can be informed that it is deadlocked in the following way: a process declares itself "deadlocked" to all other processes upon detecting deadlock. Any process which is waiting only for deadlocked processes also declares itself "deadlocked." If it is necessary for every deadlocked process to know what the other deadlocked processes are waiting for, then this information may also be propagated.

We can ensure that no process has a backlog of an unbounded number of queries by requiring a process to receive acknowledgements to earlier queries before sending the next query.

Our algorithm requires that all processes which are permanently idle (including terminated, failed or aborted processes) reply to queries. If failure prevents such a reply from being sent, the failure must be detected by other means and the reply sent.

ACKNOWLEDGEMENT

We are particulary grateful to E. W. Dijkstra and C. S. Scholten for their encouragement and advice. In particular, the proof of Theorem 2 was suggested by Scholten. We were helped greatly by comments from C. A. R. Hoare, N. Francez, G. Andrews and F. Schneider.

References

[1] Chang, E. "Decentralized Deadlock Detection in Distributed Systems," University of Toronto.

[2] Dijkstra, E. W. and C. S. Scholten. "Termination Detection for Diffusing Computations," Information Processing Letters 11, 1, August 1980, pp 1-4.

[3] Gligor, V. D. And S. H. Shattuck. "Deadlock Detection in Distirbuted Systems," IEEE Transactions on Software Engineering, SE-6, 5, September 1980, pp 435-440.

[4] Goldman, B. "Deadlock Detection in Computer Networks," Tech. Rept. MIT-LCS-TR185, M.I.T., September, 1977.

[5] Gray, Jim. "Notes on Database Operating Systems," in Lecture Notes in Computer Science, Springer Verlag, 1978.

[6] Haas, L. M. Two Approaches to Deadlock in Distributed Systems, Ph.D. Thesis, Computer Science Department, University of Texas at Austin, July 1981.

[7] Hoare, C. A. R. "Communicating Sequential Processes," Comm. ACM 21, 8, pp 666-677, August 1978.

[8] Isloor, S. S. and T. A. Marsland. "An Effective 'on-line' Deadlock Detection Technique for Distributed Data Base Management Systems," Proceedings COMSAC 1978, IEEE, pp 283-288, 1978.

[9] Lomet, D. B. "Coping with Deadlock in Distributed Systems," Research Report RC 7460 (#32196), IBM T. J. Watson Research Center, December 1978.

[10] Mahoud, S. A. and J. S. Riordon. "Software Controlled Access to Distributed Data Base," INFOR 15, 1, pp 22-36, February 1977.

[11] Menasce, D. and R. Muntz. "Locking and Deadlock Detection in Distributed Databases," IEEE Transactions on Software Engineering, SE-5, 3, pp 195-202, May 1979.

[12] Misra, J. and Chandy, K. M. "Termination Detection of Diffusing Computations in Communicating Sequential Processes," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, January 1982.

[13] Obermarck, R. "Global Deadlock Detection Algorithm," IBM San Jose Research Laboratory, June 1980.

[14] Lamport, L. "Time, Clocks, and the Ordering of Events in a a Distributed System," Communications of the ACM, No. 7, July 1978.

[15] Rationale for the Design of the ADA Programming Language, ACM Sigplan Notices, Vol. 14, No. 6, June 1979.

Appendix 1 An Example of the Operation of the Algorithm of Section 3

Consider a network consisting of 4 processes. Its initial state and a possible computation sequence is given below.

Initial states of processes:

| Process: | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| State: | idle Waiting for 2 or 3 | idle Waiting for 4 | idle Waiting for 1 or 4 | Executing |

Computation step

1: 1 sends queries to 2 and 3 (initiation)

2: 2 receives query from 1 (2 gets engaged)
2 sets its engager to 1
2 sends query to 4

3: 3 receives query from 1 (3 gets engaged)
3 sets its engager to 1
3 sends queries to 1 and 4

4: 4 sends message to 3

5: 1 receives query from 3
1 sends reply to 3 (since the initiator is always engaged)

6: 4 changes state from executing to idle and waits for 2

7: 4 receives query from 2 (4 gets engaged)
4 sets its engager to 2
4 sends query to 2

8: 4 receives query from 3
4 sends reply to 3

9: 3 receives message from 4 (sent at step 4)
3 becomes executing

10: 2 receives query from 4
2 sends reply to 4 (since 2 is already engaged)

11: 3 receives reply from 4 (sent at step 8)
But 3 has not been continuously idle since it got engaged (at step 3) hence 3 will never send its reply to 1

The reader may convince himself that process 1 will never receive all the replies (since 3 will not send a reply - see step 11). Furthermore, process 2 will send a reply to 1. Note that processes 2 and 4 are deadlocked at step 11. However, this query computation is only concerned about whether process 1 is deadlocked. (Processes 2 and 4 will have to initiate their own query computations to detect that they are deadlocked.) This has also been elaborated in Haas [6]. An algorithm for a related problem appears in Misra and Chandy [12].

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>AFOSR-TR- 82-0882 | 2. GOVT ACCESSION NO.<br>AD-A120 372 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>A DISTRIBUTED DEADLOCK DETECTION ALGORITHM AND ITS CORRECTNESS PROOF | | 5. TYPE OF REPORT & PERIOD COVERED<br>TECHNICAL<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>K.M. Chandy, J. Misra, and L. Haas | | 8. CONTRACT OR GRANT NUMBER(s)<br>AFOSR-81-0205 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Computer Sciences Department<br>University of Texas<br>Austin TX 78712 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>PE61102F; 2304/A2 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Directorate of Mathematical & Information Sciences<br>Air Force Office of Scientific Research<br>Bolling AFB DC 20332 | | 12. REPORT DATE<br>Feb 1982<br>13. NUMBER OF PAGES<br>20 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Accepted for publication in the Communications of the ACM.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a very simple distributed algorithm for deadlock detection in a network of processes. The algorithm is proven correct, i.e., the authors show that all true deadlocks are detected and no false reporting of deadlock occurs. In this algorithm no process maintains global information. All messages have identical length and are short, consisting of a single node name and a sequence number. The authors' work is based on the work of Dijkstra and Scholten on termination detection of diffusing computations.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)


ND
ATE