# DETECTING TERMINATION OF DISTRIBUTED COMPUTATIONS USING MARKERS

Jayadev Misra
Department of Computer Sciences, University of Texas, Austin, 78712

**Abstract**

A problem of considerable importance in designing computations by process networks, is detection of termination. We propose a very simple algorithm for termination detection in an arbitrary network using a single marker We show an application of this scheme in solving the problem of token loss detection and token regeneration in a token ring.

## Introduction

We study the problem of detecting termination of computation in a network of processes. If every process in a network is *idle*, i.e. waiting for messages in order to carry out further computations, and there are no messages in transit, i.e. all messages that have been sent have been received, then no process will carry out any further computation. It is often important to detect such a situation. Multiphase algorithms [15] in which a phase is to be initiated only upon completion of the previous phase, requires termination detection of a phase. Francez, Rodeh and Sintzoff [11] suggest that it may be easier to devise a distributed algorithm in two steps: (1) design an algorithm that maintains the desired safety properties and eventually guarantees a terminating *global* state; (2) superimpose a termination detection algorithm on the basic algorithm. Deadlock detection [8] which is related to termination detection is of fundamental importance in distributed data bases. Detection of token loss in a token ring can be shown to be a termination detection problem.

We suggest an algorithm for termination detection in an arbitrary network of processes. The algorithm uses a single marker which repeatedly traverses the edges of the network until it detects termination. We make no assumption about the network structure, process behavior or message delays. Our only assumption is that a process receives (only and all) messages from another process in the order sent by the sender. Our solution is symmetric among the processes, i.e. process id's are not used in the solution. As an application of the scheme in this paper, we give a simple and efficient algorithm for detecting token loss and regenerating the token in a token ring network. The marker algorithm has also been applied [14] in *avoiding* deadlocks in distributed simulations.

There has been a considerable amount of work in termination detection. For a specific class of computations, called *diffusing computations*, Dijkstra and Scholten [2] proposed a very elegant algorithm. Their approach has been extended and applied to a variety of problems [4,5,6,7]. The major drawbacks of their approach are: (1) termination detection algorithm must be initiated whenever the diffusing computation starts and (2) the number of messages used for termination detection is equal to the number of messages in the diffusing computation itself. Our solution does not suffer from these drawbacks.

A series of papers has been published by Francez and co-workers [9,11,12] leading to a marker type algorithm. The algorithm proposed in this paper refines, simplifies and removes some of the restrictions of their approach. Independently Gouda [1] and Dijkstra [3] have proposed similar schemes, again with certain restrictions. All of these schemes use the marker to determine if a process has remained "continuously idle" over an interval of time; the idea of continuous idleness also appears in [8] for deadlock detection.

Recently Chandy and Lamport [10] have proposed a very elegant and general scheme for detecting *stable properties* of a network; a property (proposition) P is stable if it remains true once it becomes true. Clearly properties such as "network computation has terminated," and "a subset of processes are deadlocked" are all stable properties. Even though we treat the narrower problem of termination detection, our solution is simpler and more efficient for this specific problem.

## Problem Description

We consider computations in finite networks of processes in which processes communicate only by messages. A process is either *idle* or *active* at any time. (For simplicity in exposition, we assume that processes do not terminate. Extension of the algorithm for

*This work was supported by a grant from IBM.

terminated processes is straightforward.) Only active processes can send messages. An idle process becomes active only upon receiving a message; an active process may become idle at any time. Every message sent in the network is received by its intended recipient after an arbitrary (possibly zero) delay. All messages sent by a process x to another process y will be received by y in the order sent by x. A network's computation terminates when every component process is idle and there is no message in transit, because in this case no process will ever become active again.

It is required to develop an algorithm to be superimposed on the basic computation in which, (1) termination is reported only when network's computation terminates and (2) termination is reported within finite time of the termination of network computation.

Initially, (i.e. when the termination detection algorithm is initiated), processes are in arbitrary states and there are arbitrary number of messages in transit.

### The Marker Algorithm

In the following algorithm, a marker visits all the processes in the network and checks to see if they are idle or active. Because of messages in transit, the marker *cannot* assert that the computation has terminated if it finds all processes to be idle after one round of visits. However, for the special case of a network in which processes are arranged in the form of a ring (i.e. every process has a unique predecessor from which it can receive messages and a unique successor to which it can send messages), the marker *can* assert that the computation has terminated if it finds after one round of visits that every process has remained *continuously idle* since the last visit of the marker to that process. This algorithm can be implemented quite easily. The marker is a special type of message, sent only by an idle process. The marker paints a process *white* when it leaves the process. A process turns *black* if it becomes active. If the marker arrives at a white process, it can claim that the process has remained continuously idle since the marker's last visit. The marker detects termination if it visits N white processes in a row, where N is the number of processes in the ring.

We generalize this scheme for arbitrary networks of processes. Crucial to our technique is the assumption that messages sent by one process to another are received in the order sent; therefore if the marker is sent from process x to process y (at which point process x is white) and on a subsequent visit the marker finds process x to be white then it can assert that (1) x has remained continuously idle during this interval and (2) there is no message in transit along edge (x,y) at this point because all messages along (x,y) would have been "flushed out" when the marker was received by y and subsequently x could not have sent a message since it has remained idle. Therefore it is necessary for our marker to traverse every edge of the network to guarantee that there are no messages in transit. For the moment, we assume that the network is strongly connected; extension of this scheme for arbitrary networks is sketched in the next section.

In every strongly connected network there exists a cycle c, not necessarily a simple cycle, which includes every edge of the network at least once. Let c denote the length of c. The marker will carry an integer m with it with the meaning that all processes seen during the last m edge traversals have been continuously idle, i.e. each of them was white when the marker arrived at the process. The entire algorithm is defined by the following rules.

**(RO)** Initially, every process is black. Marker departs from an arbitrary process x along some outgoing edge, according to rule (R1) below.

**(R1)** Marker departure from process x: Marker departs along the next edge (x,y) of c only when x is idle. Prior to departure set,

$$m := \begin{cases} 0, & \text{if x is black} \\ m+1, & \text{if x is white} \end{cases}$$

{termination is reported if m=c}

and paint x white.

{To guarantee that termination is eventually detected, we require that the marker cannot stay permanently at an idle process.}

**(R2)** Message arrival at process x: process x paints itself black.

### Proof of the Algorithm

We must show that (1) if the network's computation terminates then the marker eventually declares termination and (2) if the marker declares termination, then network computation has terminated. Property (1) is easy to see: Rule (R1) will be repeatedly applied upon termination and Rule (R2) will never be applied; therefore every process will eventually become white and after that m will increase to c. We next prove Property (2). The following proof, due to Chandy, considerably simplifies the original proof of the author.

If the marker reports termination (m=c), consider the time instant t at which the marker last set m to zero and departed from a process. We show that the network computation must have terminated at t. Since the marker eventually sets m to c, it traverses every edge of the network and visits every process, after t. Since it never sets m to zero after t, every process must be white when the marker visits it (after t) and therefore every process must be white at t. There can be no message in transit along any edge at t; if there were a message in transit along edge (x,y), then when the marker traverses the edge (x,y) and arrives at y, it must find y black since y must have received the message before the marker.

### Extension for Arbitrary Networks

**(1)** This algorithm can be applied to a network G which may not be strongly connected. Let $G_1, G_2$ ... be

the maximal strongly connected components of G. Define $G_i$ to be a *predecessor* of $G_j$ if processes in $G_j$ are reachable from processes in $G_i$. The algorithm is applied successively to each strongly connected component starting with components which have no predecessors and the marker moving to a new component only after termination has been detected for all its predecessors. Therefore the marker visits the components in a topological sort sequence.

We claim that every component for which termination has been detected will remain terminated. We prove it using induction on the topological sort sequence of the components.

(i) All components without predecessors must remain terminated after termination detection because they cannot receive messages from predecessors and become active.

(ii) Assume that every predecessor of $G_j$ for which termination has been reported, will remain terminated. When termination is reported for $G_j$, $G_j$ will remain terminated because from induction hypothesis, no process in $G_j$ will ever receive a message from any predecessor component and thereby become active.

The network computation terminates when every strongly connected component has been declared terminated.

**(2)** One drawback of the algorithm is the requirement that the network be preprocessed to determine its maximal strongly connected components and a cycle for each component containing all edges. We sketch a distributed algorithm, i.e. an algorithm in which no global information is available to any process. We assume that the marker can traverse edges in both directions. Then as far as the marker is concerned, the network is connected and undirected (and hence strongly connected).

We may use any standard search strategy for the marker to traverse all edges of this network. We sketch the algorithm with depth first search as the strategy. A new depth first search (we call it a *round*) is started when a black process is seen. The root of a round is the (black) process where the depth first search started. The marker carries with it the number of its current round; each process retains the last round number of the marker, when the marker visits the process. The marker increments its round number (starts a new round) upon arriving at a black process. Then it paints the process white and departs along some edge when the process is idle. The algorithm for a process that is white (prior to marker departure) is given below.

*if the marker has a different (higher) round number:*
*{join the depth first search}*

```
designate the sender of the marker as the
father;
update one round number;
propagate the marker (see below)
```

*if the marker has the same round number:*

```
if the marker has come from a son
   then propagate the number (see below)
   else {marker has come from a process
         other than a son}

return the marker to the last sender.
```

Propagate the marker:

choose some edge along which the marker has not been received or sent in the current round;

```
if there is such an edge
   then send the marker along that edge
   else {there is no such edge} send marker
        to the father or if there is no
        father {this is the root process}
        report termination.
```

Proof of the algorithm is identical to the previous proof; note that the depth first search of an undirected graph leads to the construction of a cycle which includes all the edges (in both directions).

**Resilient Token Ring**

A token ring provides a useful mechanism for ensuring that at ay time at most one process out of a group of processes can enter a critical section. The "marker" in the termination detection algorithm is a special case of a token. The group of processes $p_0,p_1,\cdots,p_{N-1}$ are arranged in a ring where $p_i$ receives messages from $p_{i-1}$ and sends messages to $p_{i+1}$ (additions, subtractions are modulo N). A single token circulates among the processes in the order $p_0,p_1,\cdots,p_{N-1},p_0,\ldots$ etc. A process $p_i$ which wishes to enter its critical section waits until it receives the token from $p_{i-1}$, then it enters its critical section and sends the token to $p_{i+1}$ upon completion of the critical section. A process which does not wish to enter its critical section merely transmits the token, perhaps after finite delay, upon receiving it. It follows then that two processes cannot enter critical sections simultaneously since at most one of them can hold the token at any time. Also if the execution of critical section always terminates, every process will send out the token within finite time of receiving it. Hence a process will execute its critical section within finite time of wishing to do so.

A problem in connection with token rings is recovery upon loss of the token. In this case, the token loss must be detected and the token must be regenerated.

A solution for both token loss detection and token regeneration has been suggested by Le Lann [13]. In this solution, every process waits a certain period of time to receive the token. If the token is not received

within this time interval, the process assumes that the token is lost and initiates the token regeneration process. Since there is a possibility that the original token may not have been lost or too many new tokens may be generated, a scheme is proposed for destroying additional tokens. This scheme makes use of process id's which are assumed to be distinct positive integers.

### A New Solution to the Token Regeneration Problem

Our solution is based on the observation that loss of the token amounts to termination of the network computation (we ignore messages other than the token in defining network computation), because all processes are idle and there is no message in transit. We use the marker algorithm to detect termination and regenerate the token. However we exploit the structure (token ring) to arrive at a solution which is somewhat simpler. Since a marker resembles a token, we propose to circulate *2 tokens*, token A and token B, in the token ring. One of these will be designated the primary token to be used by processes for entering their critical sections. However for token loss detection and token regeneration the two tokens are treated symmetrically, i.e. any token may be used to detect the loss of the other token and regenerate it. Our solution does not use time-outs nor does it make use of process id's. Therefore, it may use fewer messages. Furthermore, it is simple enough that it can even be implemented in hardware; in particular it avoids high-low comparisons and uses only equal-unequal comparisons between integers. We will show that the loss of a token, say token A, will be detected by token B within one round of token B's travel. Therefore the algorithm with 2 tokens is guaranteed to work if no token fails within a round (of travel) of the other token's loss. The algorithm is easily extended to k tokens, for any $k>1$; in that case the algorithm will work if at least one token is guaranteed to make a round after the loss of one or more tokens.

Our solution makes use of the following observations: a token at process $p_i$ can guarantee that the other token is lost if since this token's last visit to $p_i$, neither this token nor $p_i$ have seen the other token. We next show how to incorporate this observation into a simple algorithm.

We associate with each token a number: nA,nB denote the numbers of tokens A and B. Each process $p_i$ also carries a number $m_i$ which is the associated number of the token last seen by $p_i$. nA,nB are both updated whenever the two tokens encounter each other.

Initially ::

nA:=1; nB:=-1; all $m_i$'s are zero.

When process $p_i$ receives token A {analogous algorithm for token b}::

```
if m_i = nA

then {token B is lost:  token A has made
      a complete round without changing
      nA; token B has not visited this
      process p_i in the mean time}

      regenerate token B

else {token B is not lost} m_i := nA;
```

When tokens encounter each other::

```
nA := nA + 1; nB := nB - 1
```

Process $p_i$ regenerates token B {analogous algorithm) for regenerating token A}::

```
nA := nA + 1; nB := -nA;
{process p_i now holds both tokens}
```

The algorithm may cause nA to become arbitrarily large (or nB to become arbitrarily small). This can be avoided as follows: we observe that the algorithm only requires nA,nB to take on values different from all $m_i$'s when they are updated (this is a consequence of equal-unequal comparison employed in the algorithm). This can be achieved by incrementing nA by 1, modulo (N+1) (analogously for nB). We claim that nA will never get the same value after incrementation as some $m_i$; if it does then nA must have been incremented at least (N+1) times following its last visit to $p_i$. This is not possible since there are N processes and one token encounters the other token at most once at a process.

We leave the extension for k tokens to the reader. By choosing a suitably large k, the probability of loss of all tokens can be made arbitrarily small.


### Acknowledgement

I am indebted to Professors M. Gouda, E. W. Dijkstra and Nissim Francez who brought their works to my attention. Professor Chandy's help in formulating the proof of the marker algorithm is greatly appreciated.


### References


1. Gouda, M. "Personal Communication," Department of Computer Sciences, University of Texas, Austin, Texas 78712.

2. Dijkstra, E. W. and Scholten, C. S., "Termination Detection for Diffusing Computations," *Information Processing Letters*, 11, 1 (Aug. 1980), pp. 1-4.

3. Dijkstra, E. W., "Distributed Termination Detection Revisited," EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

4. Cohen, S. and Lehmann, D., "Dynamic Systems and Their Distributed Termination," *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 18-20, 1982, pp. 29-33.

5. Misra, J. and Chandy, K. M., "A Distributed Graph Algorithm: Knot Detection," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, October 1982, pp. 678-686.

6. Chandy, K. M. and Misra, J., "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM*, Vol. 25, No. 11, November 1982, pp. 833-837.

7. 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, pp. 37-43.

8. Chandy, K. M., Misra, J., and Haas, L., "Distributed Deadlock Detection," *ACM Transactions on Computing Systems*, Vol. 1, No. 2, May 1983, pp. 144-156.

9. Francez, N. "Distributed Termination," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, January 1980, pp. 42-55.

10. Chandy, K. M. and Lamport, Leslie, "Detecting Stability in Distributed Systems," in preparation.

11. Francez, N., Rodeh, M. and Sintzoff, M., "Distributed Termination with Interval Assertions," *Proceedings of Formalization of Programming Concepts*, Peninsula, Spain, April 1981, Lecture Notes in Computer Science 107, (Springer-Verlag).

12. Francez, N. and Rodeh, M., "Achieving Distributed Termination Without Freezing," *IEEE-TSE*, Vol. SE-8, No. 3, May 1982, pp. 287-292.

13. Le Lann, Gerard, "Distributed Systems - Towards a Formal Approach," *Information Processing 77*, B. Gilchrist, Editor, IFIP, North-Holland Publishing Company (1977).

14. Kumar, Devendra, Ph.D. Thesis (in preparation), Computer Sciences Department, University of Texas, Austin, 78712.

15. Chandy, K. M. and Misra, J., "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, Vol. 24, No. 4, April 1981, pp. 198-205.