A CLASS OF TERMINATION DETECTION ALGORITHMS FOR DISTRIBUTED COMPUTATIONS[1]

Devendra Kumar

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

TR-85-07 May 1985

[1]This work was supported by Air Force Grant AFOSR 81-0205.

Abstract

We present a class of efficient algorithms for termination detection in a distributed system. These algorithms do not require the FIFO property for the communication channels. Assumptions regarding the connectivity of the processes are simple. Messages for termination detection are processed and sent out from a process only when it is idle. Thus it is expected that these messages would not interfere much with the underlying computation, i.e., the computation not related to termination detection. The messages have a fixed, short length. After termination has occurred, it is detected within a small number of message communications.

The algorithms use markers for termination detection. By varying assumptions regarding connectivity of the processes, and the number of markers used, a spectrum of algorithms can be derived, changing their character from a distributed one to a centralized one. The number of message communications required to detect termination after its occurrence depends on the particular algorithm – under reasonable connectivity assumptions it varies from order N (where N is the number of processes) to a constant.

This paper introduces message counting as a novel and effective technique in designing termination detection algorithms. The algorithms are incrementally derived, i.e., a succession of algorithms are presented leading to the final algorithms. Proofs of correctness are presented. We compare our algorithms with other work on termination detection.

Table of Contents

1. Introduction

We develop a class of efficient algorithms for termination detection in a distributed system. We do not require the FIFO property for the communication channels, which is usually assumed in other works. (The FIFO property for a communication channel means that messages in the channel are received in the same order as they were sent.) Our assumptions regarding connectivity of processes are simple. We have categorized our algorithms in three classes. Algorithms in classes 1 and 2 assume that there exists a cycle involving all processes in the network. This cycle need not be an elementary cycle, i.e., a process may be arrived at several times in a traversal of the cycle. Moreover, the edges of the cycle need not be *primary edges*, i.e., the edges involved in the underlying computation; *secondary edges* may be introduced in the network to facilitate termination detection. (We use the terms *edges*, *lines*, and *channels* interchangeably.) Normally the length of this cycle would affect performance of the algorithms; by using secondary edges, if necessary, the length of this cycle may be kept to a minimum. Algorithms in class 3 assume the existence of cycles in several parts of the networks.

In these algorithms, messages for termination detection are processed and sent out from a process only when it is idle. Thus it is expected that these messages would not interfere much with the underlying network computation, i.e., the computation whose termination is to be detected.

Except for algorithms in class 1, the messages for termination detection in these algorithms have a fixed, short length (a pair of integers). In all algorithms presented, termination is detected within a small number of message communications after its occurrence.

In devising an algorithm for detecting termination, deadlock, or some other stable property [Chandy 85a, Chandy 85b], one important issue is how to determine if there are no primary messages in transit (*primary messages* are those transmitted in the underlying computation; *secondary messages* are those related to termination detection). Several approaches have been developed to handle this issue – acknowledgement messages [Chandy 85a], using a marker to "flush out" any messages in transit (with the assumption of FIFO property) [Misra 83, Chandy 85a], etc. One contribution of this paper is to suggest a new approach – counting the number of primary messages sent and received. As shown in this paper, this approach has several desirable features – it results in simple and flexible connectivity requirements, it does not require the FIFO property for the communication channels, and it does not generate too much overhead in terms of the number of secondary messages after the occurrence of termination. Moreover, we show that it is not necessary to count and transmit information regarding number of primary messages on *individual lines* – it is sufficient to count and transmit information about the total number of primary messages received and the total number of primary messages sent by individual processes.

Classification of Our Algorithms

Algorithms in class 1 are based on counting primary messages on every line. Each process keeps a count of the number of primary messages it has received or sent on each adjacent line (i.e., input line or output line respectively). As mentioned above, algorithms in class 1 assume that there exists a cycle C including every process of the network at least once. A marker traverses the cycle, and uses these counts in detecting termination. After termination has occurred, it will be detected within $|C|$-1 communications of the marker. ($|C|$ refers to the length of the cycle C, i.e., the number of edge traversals required to complete the cycle.) The problem with this algorithm is that each message is long – it consists of E number of integers where E is the total number of primary lines in the network.

Algorithms in class 2 reduce the message length. In these algorithms, each process counts the total number of primary messages received by it, and the total number of primary messages sent by it. Here counts are not being kept for individual adjacent lines. A marker traverses the cycle C, and collects this information to detect termination. In this case the message length is short (two integers). After the occurrence of termination, it will be detected within $2 \cdot |C| - 2$ message communications. Note that if C is an elementary cycle then $|C| = N$, where N is the number of processes in the distributed system.

Next, class 3 of our algorithms improve the performance of the algorithms in class 2, by using multiple markers which traverse different parts of the system. We make simple connectivity assumptions to permit these traversals. Using two markers, under reasonable assumptions the number of message communications after the occurrence of termination is reduced to approximately 3N/2, each message carrying an integer and a boolean. As the number of markers is increased, this number reduces further and the algorithm tends to change its character to a centralized one. Finally, using N markers, this number is reduced to the constant 4, and the algorithm becomes a purely centralized one.

On the Nature of This Presentation

A number of excellent papers on deadlock and termination detection for distributed systems have appeared in recent years. These papers usually discuss how the algorithm executes, i.e, what are the key data or execution steps in the algorithms. Proofs of correctness are usually provided to convince the reader that the given algorithm works. However, certain other important questions are usually left unanswered. How was the algorithm developed in the first place? Why were certain decisions (conventions, assumptions, major data, major execution steps) in the design of the algorithm taken – are they critical to the correctness, or are they present simply to enhance performance, or understandability, etc.? How would a simple variation of these decisions affect either correctness or performance? For the algorithms discussed here, our presentation attempts to answer some of these questions to a certain extent. We discuss a succession of algorithms, each algorithm differing from the previous ones in a simple manner. Several simple variations of the algorithms are considered. As would be noted in the discussion, some of these "algorithms" are not even correct; they are discussed simply to enhance understandability of later algorithms. Moreover, specific details, for instance initial conditions, are derived from more general considerations. It is hoped that with this method of presentation, the reader can develop a better insight as to how various decisions were arrived at. Since the relationships among various algorithms are explicitly discussed, this approach would also help keep a clear and organized view of the class of algorithms presented.

Related Work

Termination detection in distributed systems has been a subject of much study in recent years. One of the earliest works in this area is the elegant algorithm of [Dijkstra 80]. This is one of the few algorithms that do not assume the FIFO property for the communication channels. However, this algorithm requires that for any primary line from a process i to a process j, there must be a line from j to i. Termination is detected within N message communications after its occurrence, where N is the number of processes in the system. However, depending on the nature of the underlying computation, *in the entire computation* the total number of secondary messages generated in this algorithm may be too much. (The total number of secondary messages in this algorithm is equal to the total number of primary messages.) This may severely affect performance. Moreover, secondary messages are processed and sent out from a process even when it is active. This may slow down the underlying computation itself.

The above algorithm was extended in [Misra 82a] to CSP [Hoare 78] environment. The basic idea of the algorithm has been used in several distributed algorithms in many application areas [Cohen 82, Misra 82b, Chandy 82b, Chandy 81].

Marker based algorithms usually do not suffer from the drawbacks mentioned above for the algorithm in [Dijkstra 80]. A marker is sent from a process only when it is idle. Therefore normally the secondary computation would not significantly slow down the underlying computation. (*Secondary computation* is that related to termination detection; the underlying computation is also called the *primary computation*.) Moreover, usually the total number of secondary messages would also be small. Roughly speaking, if the primary computation becomes more intense (i.e., primary messages are being generated at a higher rate), then the recipient processes are likely to be active more of the time (i.e., idle for lesser time). Hence the marker is likely to move less frequently since it has to wait till the process has become idle. However, this is not to say that marker based algorithms always result in better performance; in fact many such algorithms require more than N message communications after the occurrence of termination.

Distributed termination detection using a marker was devised by Francez et. al. [Francez 80, Francez 81, Francez 82]. This approach was improved upon, removing some of its restrictions, in another marker based algorithm [Misra 83]. In this algorithm a marker traverses a cycle C' that includes every edge of the network at least once. The algorithm requires the FIFO property for the communication channels. Termination is detected, after its occurrence, within two rounds of this cycle. Note that, in principle, assuming the existence of a cycle traversing every edge is equivalent to assuming the existence of a cycle traversing every process (as in our approach). However, the performance resulting from the two approaches would normally be different. The cycle C' in general may be quite large – usually it would be longer than the total number of primary edges in the network, and the number of primary edges can be $O(N^2)$. In contrast, in our approach we can always define an elementary cycle (whose length will be N), introducing secondary edges if necessary. Defining an optimal or near optimal cycle in our approach is much simpler, since we don't require the cycle to involve every primary edge. If the network is evolving over time (e.g., new primary lines or processes being added to the network) our approach would normally require simpler changes in the data stored at the processes regarding this cycle.

In several recent works [Chandy 85a, Chandy 85b) the notion of termination and deadlock has been generalized and elegant schemes have been presented to solve these general problems. [Chandy 85a] shows how the general scheme presented there can be applied in many ways to solve the specific problems of termination and deadlock detection. The termination detection algorithm described there assumes the FIFO property for the communication channels. A marker traverses a cycle that includes every process of the network at least once. Termination is detected, after its occurrence, within two rounds of this cycle. The marker is a short message, containing only one integer. However, before the marker is sent out from a process, another message (containing no data) is sent out on output lines of this process. This effectively doubles the number of message communications after occurrence of termination. Since our algorithms in class 2 involve two rounds of the same cycle, with each secondary message having two integers, we expect comparable performance between the above algorithm and our algorithms in class 2. However, our schemes in class 3 improve the performance even further. As indicated in [Chandy 85a], the FIFO requirement for the communication channels may be removed, leading to another algorithm. But that algorithm would involve too many acknowledgement messages (equal to the number of primary messages).

One nice property that the two algorithms above ([Misra 83] and the algorithm in [Chandy 85a] using the FIFO property) enjoy is that the termination detection algorithm may be initiated with the underlying computation of the network in an arbitrary state, i.e., there may be an arbitrary number of primary messages in transit and the processes may be in arbitrary states. Our algorithms and most of the other algorithms published require special initializations for the secondary computation before the underlying computation starts.

As mentioned earlier, termination detection has been used in designing several other distributed algorithms. Many distributed algorithms can be devised as multiphase algorithms, where a new phase is started after the termination detection of the previous phase. Distributed simulation schemes have been devised using this approach [Chandy 81, Kumar 85]. [Francez 81] suggests a methodology for devising distributed programs using termination detection.

A problem of considerable importance that is closely related to termination detection, is the problem of deadlock detection in distributed systems. Several important pieces of works have appeared in this area [Gligor 80, Beeri 81, Obermarck 82, Chandy 82a, Chandy 83, Bracha 83, Haas 83].

Synopsis of the Rest of the Paper

Section 2 defines the model of computation and defines the termination detection problem. Criteria used for comparing termination detection algorithms may vary widely – performance, storage requirements, communication cost, simplicity of implementation, etc. In this paper we concern ourselves only with performance. In section 3 we discuss our performance criteria. Sections 4, 5, and 6 discuss our algorithms in classes 1, 2, and 3 respectively (we have commented on these classes earlier in the introduction). Finally section 7 gives concluding remarks.

2. Problem Definition

First we describe a basic model of a distributed system. For ease of exposition, we discuss our algorithms in terms of this basic model. Our algorithms are applicable to more general distributed systems; we briefly mention these systems later in this section.

The Basic Model

A distributed system consists of a finite set of processes, and a set of unidirectional *communication channels* (or *lines*, or *edges*). Each communication channel connects two distinct processes. Given two processes i and j, there is at most one communication channel from i to j, denoted by the ordered pair (i, j).

In addition to their local computations, processes may send or receive messages. Process i can send a message to process j only if the line (i, j) exists. Process i does so by depositing the message in the channel (i, j). This message arrives at process j after an arbitrary but finite (possibly zero) delay. Process j receives the message by removing it from the channel, within a finite time (possibly zero) of its arrival. The channels are error-free, except that they need not be FIFO channels.

The "Underlying" Computation

Now we describe the nature of the computation (called the *underlying computation* or the *primary computation*) whose termination is to be detected. The messages sent or received in this computation are called *primary messages*. Later other computation (called *secondary computation*) would be superimposed on this computation for the purpose of termination detection.

From the point of view of the underlying computation, at any moment a process is in one of two states:

1. *Active state:* In this state, a process may send primary messages on its outgoing lines. It may become idle at any time.

2. *Idle state:* In this state, a process can not send any primary messages. On receiving a primary message, it may remain idle or switch its state to active.

A process in any of the two states may receive primary messages or do any local computations. It is assumed that initially, (i.e., when the primary computation starts) there are no primary messages in transit; though the processes may be in arbitrary states.

The Termination Detection Problem

A message in the distributed system is said to be a *transient message* if it has been sent, but has not been received yet. We say that at a moment t the distributed computation is terminated iff:

1. all the processes are idle at time t, and

2. there are no transient primary messages at time t.

It is obvious that if the network computation is terminated at a time instant t, then it would remain terminated for all times after t (unless forced otherwise by some outside agent). The problem is to detect the state of termination within a finite time after its occurrence. To this end, we will devise an algorithm to be superimposed on the underlying computation; this algorithm must satisfy the following properties:

1. Termination is reported, to some process in the network, within a finite time after termination of the underlying computation, and

2. if termination is reported at some time t, the network must be terminated at time t (i.e., no "false detection" of termination is allowed).

Messages related to termination detection are called *secondary messages*. It may be noted that an idle process may send secondary messages, even though it can not send primary ones.

Other Models

We briefly mention here other features that could be incorporated in our model of computation, without affecting the applicability of our algorithms (possibly with some minor modifications).

1. We may allow multiple communication channels from a process i to a process j. Also, a process could be allowed to send a message to itself. These extensions may be useful if a process consists of a set of interacting subprocesses.

2. A process may *broadcast* a message to a set of processes. This is equivalent to sending the same message via communication lines to each process in the set.

3. There may be a third state for a process – a *terminated state*. A process enters this state when it is guaranteed that it will not send out any primary messages in future, and no more primary messages would arrive at its input lines.

3. Performance Criteria

There are two major criteria for performance evaluation of termination detection algorithms:

1. The effect of secondary computation on the primary computation itself, i.e., how the primary computation gets slowed down and

2. How long it takes to detect termination after its occurrence.

In general, the two criteria above would be assigned different weights, depending on the objectives of the primary computation and its termination detection. One has to consider not only the time delays involved, but also how *time critical* the two delays are. Depending on application, one of these may carry a higher weight than the other. The following examples illustrate this:

1. Consider a distributed system that monitors a physical system. The primary computation is triggered by an extreme state in the physical system and its objective is to bring the system to a steady state. The primary computation terminates after the system returns to the steady state. Here the former criterion would be more significant.

2. Consider a secondary computation whose objective is to detect the termination of a token in a token ring [Misra 83]. Suppose the loss of the token represents an extreme state that must be corrected immediately. Here the second criterion would be more significant.

3. Consider a multiphase distributed simulation [Chandy 81, Kumar 85]. Here

the objective is to reduce the *total* simulation time. In this case none of the two delays above are time critical and both affect the overall objective in the same way; thus both criteria would have equal weights here.

In this paper we will focus on the second criterion. (As mentioned above, in a particular application this may or may not be a good criterion for performance measurement.) Let I denote the time interval between the occurrence of termination and its detection.

How should one estimate I? Obviously, the value of I depends on characteristics of the system that supports the primary computation. We use the number of (secondary) message communications during the interval I and the lengths of these messages as a measure of I. Knowing the characteristics of communication delays, one may establish either I or an upper bound on it. For simplicity of discussion, we assume that any communication delay in the system is a linear function of message length. We mention below a few details about our performance evaluation:

1. Note that the value of I (and the associated measures mentioned above) would depend on where the marker is at the time when termination occurs, etc. For simplicity, we would normally consider only the worst case values.

2. Message communications at the same time on different lines will be taking place *in parallel* – this must be taken into account in determining the number of messages, i.e., during any overlapping period, only one message is considered being communicated. In general, any two independent events will be assumed to take place in parallel.

3. During the interval I, the number of messages received may be different (slightly) from the number of messages sent. We consider the latter one as the number of message communications. (This would be more reasonable in situations where the time involved in the act of sending a message, i.e., the *transmission time*, is longer than the propagation delay of the message.)

4. By message length we mean the total length of data in it. It is assumed that even for a message of length zero, there would be a non-zero communication delay.

4. Class 1 of Algorithms: Counting Primary Messages on Each Line

In these algorithms, a marker traverses a cycle C that includes every process of the network at least once (discussed in section 1). Information as to how many messages are in transit is kept by counting the number of primary messages sent, and received, for each line. Each process i has two local arrays $SNTP_i$ and $RECP_i$. (For simplicity of discussion, we assume here that primary lines in the network are globally numbered 1, 2, ..., E and each array $SNTP_i$ and $RECP_i$ has E elements. We will discuss more appropriate data structures later.) At any time, $SNTP_i(e)$ = the number of primary messages sent by process i on line e after the last visit of marker at i (or since the initial time, if the marker has not visited i yet). $RECP_i(e)$ is similarly defined for messages received. Each process i increments $SNTP_i(e)$ or $RECP_i(e)$, respectively, on sending or receiving a primary message on line e.

The marker has two arrays *SNTM* and *RECM*, where it keeps its knowledge as to how many primary messages have been sent or received on each line.

(For convenience, in this paper we use the obvious notation for *array assignments*, *array equality*, etc. Also, we often use a time argument in a variable to refer to its value at that time.)

An Algorithm-Skeleton

The following basic algorithm-skeleton is followed by the marker.

```
(* marker arrives at process i, i.e. it is received by i. *)
    The marker waits till process i becomes idle;

(* Process i is idle now. Marker starts its visit at i. *)
    $SNTM := SNTM + SNTP_i$;
    $SNTP_i := 0$;
    $RECM := RECM + RECP_i$;
    $RECP_i := 0$;
(* The visit at process i is completed. *)

(* Declare termination or depart from process i. *)
    Under an appropriate condition (to be discussed) the marker
    declares termination. If this condition does not hold,
```

the marker leaves process i along the next line on cycle C.

We discuss later (under the heading "some improvements and details") the algorithm and data structures required to facilitate the repeated traversal of the cycle C by the marker.

A process does not receive any messages during the interval between the start of marker's visit and its departure. In other words, the underlying computation at a process is carried out only before the marker's visit and after its departure. As mentioned earlier, the variables $SNTP_i$ and $RECP_i$ are incremented on sending or receiving (respectively) a primary message.

The variables related to termination detection are initialized before the primary computation starts. Initially, a value of zero is assigned to all elements of *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$. (This initialization will be changed later in the discussion.) Also, the marker is initially at an arbitrary process, and visits it when the process becomes idle.

The above is only a skeleton of an algorithm; we have not yet discussed when the marker declares termination. We address this issue now. Suppose the primary computation terminates at time T_f. Then within a finite time after T_f the system would reach a state where the condition $SNTM = RECM$ is true (i.e., the corresponding elements of the two arrays are equal) and would remain true forever. (After T_f this condition may become true or false several times, but definitely after one complete traversal of the cycle C by the marker it will remain true forever.) This is stated as theorem 1 below. This suggests a way of detecting termination, but we still have to avoid the possibility of detecting "false termination". Note that the condition $SNTM = RECM$ being true at a point in computation does not guarantee that termination has occurred. For example, initially this condition holds, but the system may have active processes. We ask the question – suppose in a sequence of visits along the cycle C, the marker continuously finds that $SNTM = RECM$. Can it conclude termination after a (predefined) finite number V of such visits? Theorem 2 looks at this question in a 'brute force' manner, and answers it in the affirmative with $V = 2.|C|$. Using this theorem one can complete the algorithm. Thereafter we consider the question of efficiency. Theorem 3 improves the efficiency of this algorithm by reducing V to $|C|$. Theorem 4 provides a way of reducing V to 1 if an additional condition is guaranteed before announcing termination. Later we discuss how to ensure this condition in an efficient way. (It will be observed that as we progress from theorem 2 towards theorem 4, the results become less obvious and the proofs of correctness more complex.) Let us first discuss some intermediate results that will be used in the proofs of these theorems.

For convenience, in this paper we will be implicitly using the convention that events are totally ordered, e.g., as in [Misra 81]. The events of interest are – sending a primary message, receiving a primary message, a process changing its state, and the marker arriving at a process, starting a visit, completing a visit, and departing the process. All time instants mentioned in this paper correspond to a point in the trace of events in the system, unless otherwise specified. In particular, normally no time instant refers to a moment in between the start and completion of a visit. (Otherwise many of our lemmas will become incorrect!)

Let $tsnt(e,\ t)$ = the total number of messages sent on line e up to (including) time t.

$trec(e,\ t)$ is similarly defined for messages received.

Let $r(e,\ t)$ = the number of transient messages on line e at time t.

Lemma 1: For any line e and any time t:

$$tsnt(e,\ t) = trec(e,\ t) + r(e,\ t) \tag{1}$$

and $tsnt(e,\ t) \geq trec(e,\ t)$ (2)

Proof: Follows from the definitions.

Lemma 2: For any line e and any two time instants t, t' such that $t < t'$:

$$tsnt(e,\ t) \leq tsnt(e,\ t') \tag{3}$$

and $trec(e, t) \leq trec(e, t')$ (4)

Proof: Follows from the definitions.

Lemma 3: For any line e = (i, j) and for any time t:

$$tsnt(e, t) = SNTM(e, t) + SNTP_i(e, t) \tag{5}$$

and $trec(e, t) = RECM(e, t) + RECP_j(e, t)$ (6)

Proof: The proof is by induction on the number of events in the system [Misra 81]. Initially, (5) and (6) are true. Also, each event leaves any of them invariant.

Lemma 4: For any line e = (i, j) and for any time t:

$$r(e, t) = SNTM(e, t) - RECM(e, t) + SNTP_i(e, t) - RECP_j(e, t) \tag{7}$$

Proof: Follows from (1), (5), and (6).

Lemma 5: Consider a "current" moment T in computation. For a line e = (i, j), suppose both processes i and j have been visited by the marker at least once. Let t_i and t_j, respectively, be the last times at which visits at processes i and j were completed. Then,

$$SNTM(e, T) - RECM(e, T) = tsnt(e, t_i) - trec(e, t_j) \tag{8}$$

Proof: Obviously $SNTM(e, T) = SNTM(e, t_i)$, $RECM(e, T) = RECM(e, t_j)$, $SNTP_i(e, t_i) = 0$, and $RECP_j(e, t_j) = 0$. The result follows from lemma 3.

Note: Later we will make certain changes that will make lemma 2 incorrect. However, lemmas 1, 3-5 will not be affected. Proofs of theorems 1-6 below will rest only on lemmas 1, 3-5 – they will not use lemma 2 directly.

Theorem 1: If the underlying computation terminates at a time T_f, then within a finite time after T_f the system would reach a state where the condition *SNTM* = *RECM* is true and would remain true forever thereafter (until termination is declared and possibly a new primary computation is started).

Proof: After T_f, all processes remain idle forever; therefore the marker does not wait indefinitely after its arrival at a process. Hence, within a finite time after T_f (say at a time T, $T \geq T_f$), the marker would have made a complete traversal of the cycle C, i.e., it would have visited every process at least once after time T_f (unless it has declared termination earlier). Let $T' \geq T$ be any "current" time. For any line e = (i, j) let t_i and t_j, respectively, be the last times at which processes i and j were visited. Obviously, $t_i \geq T_f$ and $t_j \geq T_f$. From lemma 5,

$$SNTM(e,\ T') - RECM(e,\ T') = tsnt(e,\ t_i) - trec(e,\ t_j)$$

But $tsnt(e,\ t_i) = tsnt(e,\ T_f)$, $trec(e,\ t_j) = trec(e,\ T_f)$, and $tsnt(e,\ T_f) = trec(e,\ T_f)$. The result follows.

$\Box$

Theorem 2: Suppose in a sequence of V = 2.|C| visits, the marker continuously finds the condition *SNTM* = *RECM* to be true after each visit in the sequence. Then at the end of this sequence it can conclude that the underlying computation has terminated.

Proof: Let T_0 be the time when the marker has completed |C| number of visits in the above sequence. We will show that at time T_0 the primary computation is terminated. Let t_{i0} be the time at which the marker completed its last visit at process i up to (including) time T_0. Also, let t_{i1} and t_i be the times at which the marker started and finished, respectively, its fist visit at process i after time T_0 (here we are considering the start and completion of a visit as two distinct events in the history of events in the system). Obviously, for all i $t_{i0} \leq T_0 < t_{i1} < t_i$.

We first show that at time T_0, for all primary lines e = (i, j), $SNTP_i(e,\ T_0) = 0$. In a similar manner it can be shown that $RECP_j(e,\ T_0) = 0$. Suppose for some

$e = (i, j)$, $SNTP_i(e, T_0) > 0$. Obviously $SNTP_i(e, t_{i1}) \geq SNTP_i(e, T_0) > 0$. Hence $SNTM(e, t_i) = SNTM(e, t_{i1}) + SNTP_i(e, t_{i1}) > SNTM(e, t_{i1})$. But $SNTM(e, t_{i1}) = RECM(e, t_{i1})$ and $RECM(e, t_{i1}) = RECM(e, t_i)$. Therefore $SNTM(e, t_i) > RECM(e, t_i)$. This contradicts the hypothesis of the theorem.

Since for every line $e = (i, j)$, $SNTP_i(e, T_0) = RECP_j(e, T_0) = 0$ and $SNTM(e, T_0) = RECM(e, T_0)$, it follows from lemma 4 that $r(e, T_0) = 0$. In other words there are no transient primary messages at time T_0.

Now we show that every process i is idle at time T_0. Obviously i is idle at time t_{i0}. Also, i did not receive any primary messages during the interval $[t_{i0}, T_0]$, otherwise we will have $RECP_i(e, T_0) > 0$ for the corresponding input line e, which will contradict the above result that $RECP_i(e, T_0) = 0$. Thus i is idle at time T_0. This completes the proof.

$\square$

Theorem 3: Theorem 2 remains valid if the requirement $V = 2.|C|$ in it is changed to $V = |C|$.

Proof: Let T_0 and T, respectively, be the times when the first and the last visits in the sequence are completed. For any process i, choose any particular visit that was completed in the interval $[T_0, T]$ and let t_{i1} and t_i, respectively, be the times at which this visit was started and finished. Claim (A) below can be shown easily (if i is the first process visited in the sequence and $t_i = T_0$, then (A) follows readily; for other cases it follows as in the proof of theorem 2):

(A) At any time t during the interval $[T_0, t_i]$, process i has $SNTP_i(e, t) = RECP_i(e, t) = 0$ for any adjacent primary line e.

Since $SNTM = RECM$ after each visit in the sequence, from (A) we conclude that:

(B) At time T_0 there are no transient primary messages, and

(C) Process i did not send or receive a primary message in the interval $[T_0, t_i]$.

Note that a process i may be active at time T_0. We will show that after time t_i, process i will never receive a primary message. Since any message in transit will be received after a finite time, this proves that there are no transient messages at time T when the above sequence of visits is completed. Moreover, since process i is idle at time t_i and does not receive any primary messages after time t_i, it will be idle at time T.

We say that a primary message is a *bad* message if it is received at a process i after time t_i. We will prove by contradiction that there can be no bad messages in the system. Suppose there are bad messages in the system. Let m be the bad message with the earliest time of reception (say t_r). Suppose m was sent on a line e = (i, j) at time t_s. Obviously, $t_r > t_s$ and $t_r > t_j$. Consider the following two cases.

Case 1: $t_s > t_i$, i.e., m was sent out after the marker's last visit at i. Then process i must have received a bad message after t_i and before t_s (hence before t_r). this contradicts the assumption that m is the bad message with the earliest time of reception.

Case 2: $t_s < t_i$. We have shown above (C) that process i does not send any primary messages in the interval $[T_0, t_i]$. Therefore m must have been sent before T_0. Hence m is in transit at time T_0. This contradicts (B) above. This completes the proof.

[]

Now we attempt to reduce further the length of the sequence of visits required with the condition *SNTM* = *RECM* before the marker can conclude termination. Note that in order to detect termination, the marker must visit every process at least once after the start of the secondary computation; since in our scheme the state (idle or active) of a process can not be deduced from the information available at the other processes. We show below in theorem 4 that if every process has been visited at least once, then the condition *SNTM* = *RECM* after visiting a process guarantees that termination has indeed occurred.

Theorem 4: Suppose, after visiting a process, the marker finds that *SNTM* = *RECM*.

Also, suppose the marker has visited every process at least once by this time. Then at this time T the underlying computation is in the terminated state.

Proof: Let t_i be the last time that the marker completed its visit at process i up to time T (i.e., $t_i \leq T$). We will show that after time t_i, process i would never receive a primary message. As argued in the proof of theorem 3, this leads to the conclusion.

With the above definition of t_i, we define *bad messages* in the same way as in the proof of theorem 3. The argument continues as before and case 1 is the same. Case 2 is different now and we consider it below.

Case 2: $t_s < t_i$, i.e., m was sent before the marker last visited process i. Since *SNTM*(e, T) = *RECM*(e, T), from lemma 5 we get *tsnt*(e, t_i) = *trec*(e, t_j). Consider the following two subcases.

Case 2.1: $t_i < t_j$. By definition of m, process i did not receive any primary messages in the interval $[t_i, t_j]$. Therefore process i did not send any primary messages in this interval. Therefore, *tsnt*(e, t_i) = *tsnt*(e, t_j). Hence *tsnt*(e, t_j) = *trec*(e, t_j). But there is at least one transient message, namely m, on line e at time t_j (since m was sent before t_i and received after t_j). This contradicts (1).

Case 2.2: $t_j < t_i$. Since *tsnt*(e, t_i) = *trec*(e, t_j), using (2) and (3) we conclude in this case that *tsnt*(e, t_i) = *tsnt*(e, t_j) = *trec*(e, t_j). In other words, no primary messages were sent on line e during $[t_j, t_i]$ and there are no transient messages on line e at time t_j. Hence m was sent before t_j and received by the time t_j. This contradicts with the definition of m.

[]

Note: The proof of case 2 will be simpler if one assumes the FIFO property for the communication channels. Informally, since m has been counted in *tsnt*(e, t_i) and has not been counted in *trec*(e, t_j), by the FIFO property we will get *tsnt*(e, t_i) > *trec*(e, t_j). Therefore we won't have to consider the cases 2.1 and 2.2.

Completion of the Algorithm

It may be noted that if the hypothesis of theorem 2 or theorem 3 is true then the hypothesis of theorem 4 is true as well, but not vice versa. Therefore the method suggested by theorem 4 would be more efficient. Hence we use theorem 4 to complete the algorithm. How would the marker decide that it has visited every process at least once? One brute force method would be to have a counter in the marker that counts how many visits have been completed. When this counter becomes $|C|$, obviously every process has been visited at least once. (Alternatively, the marker could count how many *distinct* processes it has visited, by marking a process "visited" after visiting it.)

We use a more efficient strategy – the initial values of the variables *SNTM*, *RECM*, $SNTP_i$, $RECP_i$ are assigned in a different way than mentioned earlier. This assignment guarantees the following two conditions:

1. As long as there is at least one process that has not been visited yet, the condition *SNTM* = *RECM* will remain false, i.e., at least one pair of corresponding elements in the two arrays will not match. (We will be assuming that each process has at least one adjacent primary line. Otherwise we have isolated processes in the system. If needed, such cases can be incorporated in the scheme in obvious ways.) This is stated as lemma 6 below.

2. Moreover, this assignment does not affect the correctness of theorems 1 and 4. (In fact, all of lemmas 1, 3-5 and theorem 1-4 remain valid.) This is stated as lemma 7 below.

Obviously, this strategy is more efficient since the additional counter in the marker is avoided, reducing its length. An infinite set of assignments guaranteeing the above conditions exist; here we consider one specific assignment.

Corresponding to every primary line e = (i, j), we initialize *SNTM*(e) = 1, *RECM*(e) = 0, $SNTP_i$(e) = 1, and $RECP_j$(e) = 2. The marker declares termination after a visit if it finds that *SNTM* = *RECM*. The rest of the algorithm remains the same as before. Theorems 5 and 6 below prove the correctness of the algorithm.

Lemma 6: With the above initialization, suppose after visiting a process the marker finds that *SNTM* = *RECM*. Then, the marker has visited every process at least once by this time (say T).

Proof: For any line (i, j), we show that the marker has visited both i and j by the time T. (Since every process has at least one adjacent line, this establishes the result.) Suppose, to the contrary, this is not true for a line e = (i, j). Consider the following cases.

Case 1: The marker has not visited the process j by the time T. Obviously, in this case *SNTM*(e, T) $\geq$ 1 and *RECM*(e, T) = 0. This contradicts the assumption that *SNTM* = *RECM* at time T.

Case 2: The marker has visited process j, but not i, by the time T. Obviously, in this case *SNTM*(e, T) = 1, and *RECM*(e, T) $\geq$ 2. Again, this leads to a contradiction. This completes the proof.

Lemma 7: With the new initial values lemmas 1 and 3-5, and theorems 1-4 remain valid.

Proof: Note that with the new initial values, lemmas 1 and 2 remain valid. The results (5) and (6) in lemma 3 become slightly incorrect – the corrected versions of these results are:

$$tsnt(\mathrm{e},\ \mathrm{t}) = SNTM(\mathrm{e},\ \mathrm{t}) + SNTP_{\mathrm{i}}(\mathrm{e},\ \mathrm{t}) - 2 \qquad (5')$$

$$trec(\mathrm{e},\ \mathrm{t}) = RECM(\mathrm{e},\ \mathrm{t}) + RECP_{\mathrm{j}}(\mathrm{e},\ \mathrm{t}) - 2 \qquad (6')$$

The proofs of (5') and (6') are similar to the proofs of (5) and (6) before. From (5') and (6') it follows that lemmas 4 and 5 remain valid.

The previous proofs of theorems 1-4 do not directly rest on lemma 2 or the initial values of the program variables (so long as lemmas 1 and 3-5 remain valid). Therefore their correctness is not affected. This completes the proof.

Theorem 5: If the underlying computation is terminated at a time T_f, then the marker would declare termination within a finite time after T_f.

Proof: Follows from lemma 7 and theorem 1.

∎

Theorem 6: Suppose at a moment T, the marker declares termination. Then at this moment, the underlying computation is, indeed, in the terminated state.

Proof: The theorem follows from lemmas 6 and 7 and theorem 4.

∎

Some Improvements and Details

We briefly mention below some simple performance improvements to the algorithm. We also discuss a few details related to implementation.

1. Instead of keeping the two arrays *SNTM* and *RECM* in the marker, it is sufficient to keep a single array, say *SRM*, which would equal *SNTM* - *RECM*. This would reduce the secondary message length. Also, this reduces the chances of an overflow. (Elements of arrays *SNTM* and *RECM* are non-decreasing with time.)

2. In our description of the algorithm, the arrays $SNTP_i$ and $RECP_i$ have an element for every primary line of the network. Usually a process is connected to only a few other processes; in such cases, with this data structure updating *SNTM* or *RECM* or *SRM* may be quite inefficient. It may be more efficient to assign contiguous local line ids to the adjacent lines at each process, keep elements only for the adjacent lines in arrays $SNTP_i$ and $RECP_i$, and keep an array that maps from local line ids to global line ids.

3. How does the marker determine the next line to be traversed? If C is a simple cycle, then obviously just keeping the successor's id at each process is sufficient. Otherwise, one may keep a circular list of outgoing lines at each process (a line may be repeated several times in this list) and a local pointer that points to the next line to be followed by the marker. These circular lists can be initialized by considering a single traversal of the cycle C "by hand". The pointers can be initialized by defining the starting point of the

marker on the cycle. Note that the marker itself does not carry any information about its path of traversal; otherwise the secondary messages would become even longer.

Performance of the Algorithm

In the worst case, the number of message communications after the occurrence of termination is |C|-1. C can be chosen to be an elementary cycle, in which case this equals N-1, where N is the total number of processes in the system. Each message has a length of E integers, where E is the number of primary lines in the system. If communication delays depend significantly on the length of the messages, then this would be quite inefficient. On the other hand, if the message length does not significantly affect communication delays then this scheme would give a reasonable performance. One nice feature of this scheme is that in the *best* case, the number of message communications after termination is zero. Normally marker based algorithms [Misra 83, Chandy85a] require at least one complete cycle between the occurrence of termination and its detection.

5. Class 2 of Algorithms: Counting Total Number of Primary Messages Sent and Received in the System

Our motivation for devising algorithms in this class is to reduce the length of secondary messages. Here the marker has two *scalar* variables *SNTM2* and *RECM2* where it keeps its knowledge regarding the total number of primary messages sent and received, respectively, in the system. This differs from algorithms in class 1 where information about *individual lines* was being kept. Each process i has two scalar variables $SNTP2_i$ and $RECP2_i$. At any time $SNTP2_i$ = the total number of primary messages sent out by process i after the last visit of the marker at i (or since the initial time, if the marker has not visited i yet). $RECP2_i$ is similarly defined for messages received. The algorithm-skeleton of class 1 remains the same for this class, except that the variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ are replaced by *SNTM2*, *RECM2*, $SNTP2_i$, and $RECP2_i$ respectively. These variables are initialized to be zero, before the primary computation starts. (Unlike the discussion in class 1, we won't find a need to change this initialization later.) As in class 1, a process does not receive any messages during the interval between the start of the marker's visit and its departure. As before, the variables $SNTP2_i$ and $RECP2_i$ are incremented on sending or receiving (respectively) a primary message.

Now we consider the issue of when the marker declares termination. Theorem 7 below states that if the primary computation terminates at time T_f then within a finite time after T_f the system would reach a state where the condition $SNTM2 = RECM2$ will be true and will remain true forever afterwards. As before, we have to avoid the possibility of detecting "false termination". Again we ask the question – suppose in a sequence of visits along the cycle C, the marker continuously finds that $SNTM2 = RECM2$. Can it conclude termination after a (predefined) finite number V of such visits? Unfortunately, the answer in this case is in the negative, as shown in example 1 below. Theorem 8 below gives a method to complete the algorithm. Theorem 9 considers simple variations of the method given by theorem 8. These variations reduce the computational requirements at the processes; they do not improve communication requirements. Theorem 10 improves the performance of the algorithm by reducing the number of message communications after termination. After proving theorem 10, we show that certain simple and obvious variations of theorem 10 do not work. First let us discuss some intermediate results that will be used in the proofs.

Note that the variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ of class 1 can be used as "auxiliary" or "ghost" variables in our proofs. The notion of auxiliary variables is discussed, for example, in [Owicki 76]. The use of these auxiliary variables in our proofs is not essential; we use them only to simplify our proofs by exploiting the results in section 4. We assume that these variables are initialized to be zero at the start of primary computation.

Lemma 8: Lemmas 1-5 and theorems 1-4 of section 4 remain valid for the present algorithm-skeleton (when the variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ are interpreted as auxiliary variables).

Proof: In the algorithm-skeleton of section 4, let us introduce variables *SNTM2*, *RECM2*, $SNTP2_i$, and $RECP2_i$ in the same way as they are used in the present algorithm-skeleton. Obviously, the previous results of section 4 hold for this new algorithm-skeleton. Now in this algorithm-skeleton, let us treat variables *SNTM*, *RECM*, $SNTP_i$, and $RECP_i$ as auxiliary variables. Obviously, the results would still hold.

Lemma 9: At any time t,

$$SNTM2(t) = \text{sum } \{SNTM(e,\ t), \text{ over all primary lines e}\} \quad (9)$$

$$RECM2(t) = \text{sum } \{RECM(e,\ t), \text{ over all primary lines e}\} \quad (10)$$

$$SNTP2_i(t) = \text{sum } \{SNTP_i(e,\ t), \text{ over all outgoing primary lines e of process i}\} \quad (11)$$

$$RECP2_i(t) = \text{sum } \{RECP_i(e,\ t), \text{ over all incoming primary lines e of process i}\} \quad (12)$$

Proof: Obvious, by induction on the number of events in the system.

Lemma 10: At any time t,

$$\begin{aligned} tr(t) = \ & SNTM2(t) - RECM2(t) + \text{sum } \{SNTP2_i(t), \text{ over all processes i}\} - \\ & \text{sum } \{RECP2_i(t), \text{ over all processes i}\} \end{aligned} \quad (13)$$

where $tr(t)$ = the total number of primary messages in transit at time t.

Proof: Let us take the sum of each side of (7) over e, e ranging over all primary lines in the system. The result follows from lemmas 9 and 8.

Theorem 7: Theorem 1 of section 4 remains valid if *SNTM* and *RECM* in that theorem are replaced by *SNTM2* and *RECM2* respectively.

Proof: Follows from lemma 8 and the results (9) and (10) in lemma 9.

[]

Example 1: To show that there exist computations (following the algorithm-skeleton) where in an infinite sequence of visits, the marker continuously finds that *SNTM2* = *RECM2*, and yet the primary computation never terminates.

Consider a network of 10 processes. The cycle C is the elementary cycle 1, 2, ..., 10, 1. Initially the marker is at process 1, process 5 is active, and process 10 is idle. Processes 1-4 and 6-9 never send or receive a primary message and are always idle. Consider the following sequence of events at processes 5 and 10:

1. 5 sends a primary message to 10, 10 receives it, 10 sends a primary message to 5, 5 receives it. At this point 5 becomes idle and 10 remains active.

2. The marker visits 5, and departs.

3. 10 sends a primary message to 5, 5 receives it, 5 sends a primary message to 10, 10 receives it. At this point 10 becomes idle and 5 remains active.

4. The marker visits 10, and departs.

5. The above steps 1-4 are repeated indefinitely.

Obviously, after every visit the marker will find that *SNTM2* = *RECM2*. But the primary computation would never terminate!

The above example illustrates why after a finite number of visits with *SNTM2* = *RECM2* after each visit, the marker can not in general announce termination. Roughly speaking, a process i may have sent and received messages in between two successive visits by the marker. Theorem 8 is based on this observation.

Theorem 8: Suppose in a sequence of V = |C| visits, the marker continuously finds that $SNTP2_i = RECP2_i = 0$ before each visit (except possibly the first visit in the sequence) and *SNTM2* = *RECM2* after each visit. Then, at the end of this sequence of visits it can conclude that the underlying computation has terminated.

Proof: We show by induction on the number of visits in the sequence that after each visit *SNTM* = *RECM*. The result follows by theorem 3.

Base Case: Consider the first visit. Let T_0 be the time when the first visit in the sequence is completed. Obviously, $SNTP2_i(T_0) = RECP2_i(T_0) = 0$ for each process in the system. Therefore from (11) and (12) in lemma 9, $SNTP_i(e, T_0) = RECP_j(e, T_0) = 0$ for each primary line e = (i, j). Hence from lemmas 4 and 1, $SNTM(e, T_0) \geq RECM(e, T_0)$ for any e. But $SNTM2(T_0) = RECM2(T_0)$. Therefore from (9) and (10) in lemma 9, we get $SNTM(e, T_0) = RECM(e, T_0)$ for every primary line e. Therefore $SNTM = RECM$ at time T_0.

Inductive Case: Inductively, suppose $SNTM = RECM$ after the k^{th} visit. By the hypothesis of the theorem, at the start of the $(k+1)^{st}$ visit $SNTP2_i = RECP2_i = 0$ where i is the process being visited. Using (11) and (12) in lemma 9 it follows that $SNTM = RECM$ at the end of the visit.

$\square$

Note: The above proof shows that if in a computation the hypothesis of theorem 8 is true then so is the hypothesis of theorem 3. The converse also follows, in an obvious way. Hence the two algorithms will require the same number of secondary message communications after and before the occurrence of termination. (Since the computation time in a visit in the two algorithms is different, the sequence of events in the two algorithms may be different. The above remark ignores any such differences.)

We state below some simple variations of theorem 8. These variations reduce only the processing requirements during a visit by the marker. Theorems 8 and 9 require the same number of secondary message communications after and before termination (again, this assumes that different processing requirements during a visit won't affect the sequence of events).

Theorem 9: Theorem 8 remains valid under any one of the following modifications (note: we are *not* considering here a combination of these modifications):

1. The requirement $SNTP2_i = RECP2_i = 0$ is replaced by $SNTP2_i = 0$.

2. The requirement $SNTP2_i = RECP2_i = 0$ is replaced by $RECP2_i = 0$.

3. The requirement "$SNTM2 = RECM2$ after each visit" is replaced by "$SNTM2 = RECM2$ at the end of the last visit of the sequence".

Proof: It is easy to see that any variation stated in theorem 9 is equivalent to theorem 8, in the sense that if the hypothesis of one is true then so is the hypothesis of the other.

$\boxed{\vert\vert}$

Now we consider a stronger modification to theorem 8. The algorithm suggested by theorem 10 below is *more* efficient than the one suggested by theorem 8, in terms of the number of message communications required after termination. We will discuss this after proving the theorem.

Theorem 10: Suppose in a sequence of $V = |C|$ visits, the marker continuously finds that $RECP2_i = 0$ before each visit (except possibly the first visit in the sequence) and $SNTM2 = RECM2$ at the completion of the last visit of the sequence. Then at the end of this sequence of visits it can be concluded that the underlying computation is terminated.

Proof: Let T_0 and T, respectively, be the times when the first and the last visits of the sequence were completed. Let t_i be the time when the marker completed its last visit at process i up to (including) time T. From (10),

$RECM2(T)$ = sum $\{RECM(e, T)$, over all primary lines e$\}$

= sum $\{trec(e, t_j)$, over all primary lines $e = (i, j)\}$ by (6).

= sum $\{trec(e, T_0)$, over all primary lines $e = (i, j)\}$ since, obviously, in the interval $[T_0, t_i]$ process i did not receive any primary messages.

Similarly,

$SNTM2(\mathrm{T})$ = sum $\{tsnt(\mathrm{e},\ \mathrm{t_i})$, over all primary lines e = (i, j)$\}$ by (9) and (5)

= sum $\{tsnt(\mathrm{e},\ \mathrm{T_0})$, over e$\}$ + sum $\{r'(\mathrm{e})$, over e$\}$

where $r'(\mathrm{e})$ = the number of primary messages sent on the line e = (i, j) during the interval $[\mathrm{T_0},\ \mathrm{t_i}]$.

Since $SNTM2(\mathrm{T}) = RECM2(\mathrm{T})$, using (2) we get $tsnt(\mathrm{e},\ \mathrm{T_0}) = trec(\mathrm{e},\ \mathrm{T_0})$ and $r'(\mathrm{e}) = 0$ for every primary line e. This is the same as conditions (B) and (C) in the proof of theorem 3. The rest of the proof is the same as the proof of theorem 3 after observation (C). (Alternatively, for any primary line e = (i, j), $tsnt(\mathrm{e},\ \mathrm{t_i}) - trec(\mathrm{e},\ \mathrm{t_j}) = tsnt(\mathrm{e},\ \mathrm{T_0}) + r'(\mathrm{e}) - trec(\mathrm{e},\ \mathrm{T_0}) = 0$. Hence by (8), $SNTM(\mathrm{e},\ \mathrm{T}) = RECM(\mathrm{e},\ \mathrm{T})$. The result follows from theorem 4.)

▯

Now we show that theorem 10 suggests a more efficient algorithm than theorem 8. Obviously, if the hypothesis of theorem 8 holds at a point in computation, then the hypothesis of theorem 10 holds as well. Example 2 below shows that the converse is not true. (However, for a given network topology, the worst case number of message communications after occurrence of termination is the same in both cases.)

Example 2: Consider the network of example 1 with the same initial conditions, except that the marker is initially at process 9. As before , processes 1-4 and 6-9 always remain idle. Consider the following sequence of events at processes 5 and 10:

1. 5 sends a message to 10, 10 receives it. At this point both processes are idle.

2. The marker arrives at process 10.

In the algorithm given by theorem 10, the marker will visit processes 10, 1, ..., 9 and then declare termination. Using the algorithm given by theorem 8, the marker will visit processes 10, ..., 5, ..., 10, ..., 4 and then declare termination.

One may be tempted to consider the following variation of theorem 10 – replace the requirement $RECP2_i = 0$ by $SNTP2_i = 0$. Example 3 below shows that this won't work.

Example 3: Consider the network of example 1 with the same initial conditions and the same behavior of processes 1-4 and 6-9. Consider the following sequence of events on processes 5 and 10:

1. 5 sends a primary message to 10 and becomes idle. (10 has not received it yet.)

2. Marker visits 5. It "restarts" a new sequence since $SNTP_5 \neq 0$ at the start of the visit.

3. Marker visits 10 and departs.

4. 10 receives the primary message sent by 5. It sends a primary message to 5 and remains active. 5 receives this message and remains idle.

5. Marker visits 5 and declares termination.

But process 10 is still active!

Since the above variation of theorem 10 doesn't work, it follows that the following variation will also not work – replace the requirement $RECP2_i = 0$ by ($RECP2_i = 0$ or $SNTP2_i = 0$). (If this variation had worked, obviously it would have been more efficient than theorem 10.)

We complete the algorithm-skeleton for class 2 by using theorem 10. Along the lines of the proof of theorem 1, it can be shown that if the primary computation terminates, say at time T_f, then the hypothesis of theorem 10 will become true within a finite time after T_f. The correctness of the algorithm follows from this observation and theorem 10.

Some Improvements and Details

1. As in class 1 (see "some improvements and details" in section 4), instead of keeping the two variables *SNTM2* and *RECM2* in the marker, it is sufficient to keep only a single variable *SRM2* which would equal *SNTM2* - *RECM2*. This has the same advantages as before.

2. Also, at a process i instead of keeping $SNTP2_i$ and $RECP2_i$, one may keep a variable $SRP2_i$ which would equal $SNTP2_i$ - $RECP2_i$, and a boolean variable to indicate if $RECP2_i = 0$. This, of course, does not improve the efficiency regarding message communications; it only reduces the processing time involved in a visit.

3. Same as 3 in our discussion under "some improvements and details" for class 1.

4. How does the marker detect that the first condition of theorem 10 holds for the entire sequence? We roughly sketch a few possible ways of doing this:

   a. <u>Sequence Length Counter:</u> The marker carries a counter for this purpose. Initially this counter is 0. On visiting a process i, if $RECP2_i$ is zero at the start of the visit then the counter is incremented; else it is reset to 1. After the visit if the counter is $\geq |C|$, then the condition $SNTM2 = RECM2$ is checked.

      What if the counter has become $\geq |C|$ and the condition $SNTM2 = RECM2$ is not met? If the counter keeps getting incremented indefinitely, it may overflow. To avoid this, one may reset the counter to 1 during a visit if it is $\geq |C|$ at the start of the visit. This raises the following issue: there seems to be a possibility that termination may be detected after "too many" visits. For example, what if the condition $RECP2_i = 0$ is true before every visit, but the condition $SNTM2 = RECM2$ becomes true after $|C|+1$ visits and the counter was reset to 1 (to avoid overflow) during the visit $|C|+1$? It can be shown that such cases can not arise. In other words, at the start of a visit if the counter is $\geq |C|$, then it can be reset to 1 without loss of efficiency. At any such point T it can be asserted that the marker will definitely visit (either in future or in current visit) a process i such that $RECP2_i \neq 0$ at the start of the visit. To prove this, suppose this is not true. Then there are two possibilities: (i) There is a finite sequence of visits made after time T such that $RECP2_i = 0$ before each such visit at the process i being visited and $SNTM2 = RECM2$ at the completion of the last visit of the sequence. (ii) There is an infinite sequence of visits made after time T such that

$RECP2_i = 0$ before each visit and $SNTM2 \neq RECM2$ after each visit. Note that after time T, for any visit at a process i if $RECP2_i = 0$ before the visit then $SNTP2_i = 0$ before the visit as well. Since $SNTM2(T) \neq RECM2(T)$, (i) above is impossible. In case (ii), obviously we have primary messages in transit at time T. Within a finite time one of these messages will be received at a process i, making $RECP2_i$ nonzero. Hence (ii) above is impossible. This completes the proof.

b. Round Number: For simplicity, first let us assume that C is an elementary cycle. The marker contains a round number. At the start of a visit (say at process i) if $RECP2_i \neq 0$ then a new round is started, i.e., marker's round number is incremented. During any visit at a process i, the marker's round number is stored in a local variable at i. If at the start of a visit, the round number of the marker equals that of the current process i, it means that the marker has previously made a sequence of at least |C| visits such that before each visit (except possibly the first one) $RECP2_j = 0$ at the corresponding process j. Therefore in this case if $RECP2_i = 0$ at the start of the visit, the condition $SNTM2 = RECM2$ is checked for termination after completion of the visit. (Alternatively, the termination check could be made at the start of a visit if the round numbers match.) At the start of secondary computation, round numbers of the marker and the processes are initialized to 1 and 0 respectively.

If the round number of the marker keeps getting incremented indefinitely, it may overflow. To solve this problem one may increment the round number as 1+ [(round number) mod |C|]. The new round number generated would obviously be different from local round numbers of all processes (except possibly the one being visited).

(As a side note, using this method the number of message communications after occurrence of termination is increased by 1.)

If C is not an elementary cycle, a counter may be kept at each process that counts the number of times the process has been visited in the current round.

c. Initial Process Id: Again let us first assume that C is an elementary cycle. In this method the marker keeps a pointer that points to the process id of the first process in the current sequence of visits such that before each visit (except possibly the first one) $RECP2_i = 0$. At the start of a visit (say at process i) if $RECP2_i \neq 0$ then this pointer is set to i. If at the start of a visit, this pointer is pointing to the current process i, this means that the marker has previously made a sequence of at least |C| visits such that before each visit (except possibly the first one) $RECP2_j = 0$ at the corresponding process j. Therefore in this case if $RECP2_i = 0$ at the start of the visit, then the condition $SNTM2 = RECM2$ is checked for termination after completion of the visit. Similar to our discussion in (a) above (using sequence length counter), if the condition $SNTM2 = RECM2$ is false in this check, we need not reset the pointer.

Obviously, there is no overflow problem in this approach. As in the method using round numbers, the number of messages after termination in this method is increased by 1. If C is not an elementary cycle, one may keep local counters at the processes to count the number of times the process currently pointed to by the marker has been visited in the current sequence.

5. Suppose we designate a specific process where the decision regarding termination would be taken. In this case the marker needs to carry only an integer (the value of *SNTM2 - RECM2*) and a boolean (instead of an integer as in 4 above) which remembers whether in the current round the first condition of theorem 10 has been true so far. Since message length has decreased, this improves the performance in the worst case. However, in the average case the number of message communications after termination will increase.

Performance of the Algorithm

The worst case occurs when the marker departs a process i and before it reaches the next process, process i receives a primary message and the primary computation terminates at this point. The number of secondary messages sent after the termination of primary computation in this case is 2.|C| - 2. Each secondary message contains two integers – one integer containing the value *SNTM2 - RECM2*, and the other used to check the first condition of theorem 10, as discussed in 4 above under "some improvements and details".

6. Class 3 of Algorithms: Using Multiple Markers

In classes 1 and 2 we have a *single* marker that sequentially traverses the system. In this section we will use multiple markers to enhance performance. First we observe that the *sequential* traversal of the system by a marker in the previous algorithms is not essential. If several processes could be visited in parallel, even then these results will hold. The following theorem is obtained from theorem 10 by an abstraction of the proof of that theorem (i.e., by avoiding details regarding sequential nature of the traversals). The proof of this theorem is essentially the same as that for theorem 10.

Theorem 11: Let $[T_0, T]$ be a time interval during which several visits have been completed, possibly in parallel. Suppose these visits satisfy the following:

1. At least one visit is completed at each process during this interval (the start times of these visits need not be in the interval).

2. At the start of each visit, say at process i, $RECP2_i = 0$, and

3. At time T $SNTM2 = RECM2$.

Then, at time T the primary computation is terminated.

Notes:

1. The values of *SNTM2* and *RECM2* at time T are defined in the obvious way – the results of various visits have to be accumulated.

2. Theorem 10 follows as a special case of theorem 11. On first sight this might not be so obvious, since theorem 10 allows the value of $RECP2_i$ for the first visit to be nonzero. However, after the very first visit in the sequence, one may consider an imaginary visit to the same process — theorem 10 would then readily follow from theorem 11.

Using theorem 11, one may devise schemes using several markers. The markers would check the values of $RECP2_i$, and accumulate values for *SNTM2* and *RECM2* in different parts of the system (these parts need not be disjoint).

A. Using Two Markers

Let us assume that we have two paths P_1 and P_2 from a given process I to a given process J. Also assume that these paths together cover all the processes in the system. Initially both markers are kept at process I. Then they traverse the two paths respectively. After both have visited J, a check for termination is made as follows. The values *SNTM2* and *RECM2* are computed by adding the corresponding values in the two markers. Each marker i also has a boolean variable $NZREC_i$ which is set to true if at the start of some visit at a process j in the current traversal of the path, $RECP2_j$ was found to be nonzero. At J, if *SNTM2* = *RECM2* and both booleans are false then termination is announced. Otherwise a new traversal is to be started. To start a new traversal, both markers may be sent back to I via a line (J, I). Alternatively, the markers may traverse the paths P_1 and P_2 in the reverse direction in which case the next check for termination would be made at process I.

Now we make a simple modification to the above scheme that would lead to an obvious generalization for the case of more than two markers. A new process, called a *central process* (*CP*), is introduced in the system where the check for termination would be made. (This process may be *implemented* as part of some existing process in the system.) Paths P_1 and P_2 now need not share their initial and final processes. Initially both markers are at the CP. A traversal of the system is started by the CP, by sending the markers to the initial processes of the respective paths. After traversing the paths, the markers arrive at the CP where the decision regarding termination is made in the same way as before.

Now we consider an erroneous variation of this scheme which supposedly attempts to improve its efficiency. Suppose a marker i has arrived at the CP after traversing its path and $NZREC_i$ is true. Suppose the other marker has not yet arrived at the CP. One might be tempted to consider the following. Since marker i knows that termination can not be announced after this traversal, it doesn't wait for the other marker to arrive; instead it goes back to traverse P_i. Equivalently, a marker i would traverse its path P_i repeatedly until the value of $NZREC_i$ is false at the end of a traversal, and then it would go to CP and wait for a termination check to be made. The following simple example shows that this scheme won't work:

Example 4: Let P_1 and P_2 consist of single processes, processes 1 and 2 respectively. Initially process 2 is idle and process 1 is active. Consider the following sequence of events:

1. Marker 2 visits process 2. It departs from process 2 (but hasn't arrived at the CP yet).

2. Process 1 sends a primary message to process 2. Process 2 receives this message and sends another one to process 1. Process 1 receives it and becomes idle. Process 2 remains active.

3. Marker 1 visits process 1. Since the value of $NZREC_1$ is true after this visit, it visits process 1 again (equivalently, after the first visit it goes to CP, then goes back and visits process 1). Now it arrives at the CP.

4. Marker 2 arrives at CP. Obviously both booleans $NZREC_i$ are false and $SNTM2 = RECM2$ at this point. Hence termination is declared. But process 2 is still active!

Performance of the Scheme

Let us assume that the length of each path P_1 or P_2 is approximately N/2. For worst case, consider the following scenario. Marker 1 visits and departs from the first process (say i) on its path. Now process i receives a primary message and at this point the primary computation is terminated. Obviously termination won't be detected after the current traversal. Again, it won't be detected in the next traversal since $RECP2_i$ would be nonzero at the start of the next visit to i (let us assume that i appears only once on P_1, and doesn't appear on P_2; otherwise this won't be strictly true). So the number of secondary message communications after termination is $\approx$ 3N/2. Each such message consists of an integer and a boolean.

B. Using More Markers

One may similarly use a CP, K paths, and K markers in general. Let L be the length of the longest of these paths. By considering a scenario similar to the above, we have the worst case number of message communications $= 3L + 4$. If each path has N/K processes then this equals 3N/K + 1. Note that as K is increased, the scheme tends to become more centralized. With K = N it is a purely centralized scheme (i.e., each process interacts only with the central processor for termination detection) with worst case number of message communications after termination = 4.

7. Conclusion

We have presented a class of efficient algorithms for termination detection in distributed systems. Our assumptions regarding the underlying computation are simple. In particular we do not require the FIFO property for the communication channels. Also, the topological requirements about communication paths are simple and flexible, both from the correctness and performance point of view. We discussed the correctness and performance of our algorithms. Depending upon the application, the nature of the chosen algorithm can be varied incrementally from a distributed one to a centralized one.

We introduced message counting as an effective technique in designing termination detection algorithms. We showed how one can avoid counting messages for *each and every line*, normally resulting in better performance. Our presentation involves deriving algorithms via a sequence of simple modifications. Several correct as well as incorrect variations have been considered. We hope that this approach of presentation has resulted in better understandability of the algorithms.

Acknowledgments

I am indebted to Professors J. Misra and K. M. Chandy for their masterful teaching of several courses at UT Austin which have significantly influenced me. Their financial support for this work is greatly appreciated. The idea of counting messages on the lines was inspired by a similar thought of Professor K. M. Chandy. I am thankful to Professor J. Misra for his encouragement and for his thoughts on performance of termination detection algorithms. Professor M. Gouda's encouragement and his positive influences on my writing style are greatly appreciated. I would like to extend my special thanks to Ted Briggs and Pradeep Jain for their comments on earlier drafts of this paper.

References

[Beeri 81] C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Research Report RJ3077*, IBM Research Laboratory, San Jose, California, May 1981.

[Bracha 83] G. Bracha and S. Toueg, "A Distributed Algorithm For Generalized Deadlock Detection", *Technical Report TR 83-558*, Cornell University, June 1983.

[Chandy 81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM*, Vol. 24, No. 4, pp.198-205, April 1981.

[Chandy 82a] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[Chandy 82b] K. M. Chandy and J. Misra, "A Computation on Graphs: Shortest Path Algorithms", *Communications of the ACM*, Vol. 25, No. 11, pp.833-837, November 1982.

[Chandy 83] K. M. Chandy, J. Misra, and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems*, Vol. 1, No. 2, pp. 144-156, May 1983.

[Chandy 85a] K. M. Chandy and J. Misra, "A Paradigm for Detecting Quiescent Properties in Distributed Computations", working paper, Department of Computer Sciences, University of Texas, Austin, Texas 78712, January 9, 1985.

[Chandy 85b] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", to appear in *ACM Transactions on Computing Systems.*

[Chang 82] E. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, pp.391-401, July 1982.

[Cohen 82] S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 29-33, Ottawa, Canada, August 18-20, 1982.

[Dijkstra 80] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol. 11, No. 1, August 1980.

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

[Francez 80] N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, pp. 42-55, January 1980.

[Francez 81] N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proceedings of Formalization of Programming Concepts*, Peninusla, Spain, April 1981. Lecture Notes in Computer Science 107, (Springer-Verlag).

[Francez 82] N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezing", *IEEE-TSE*, Vol. SE-8, No. 3, pp.287-292, May 1982.

[Gligor 80] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE-TSE*, Vol. SE-6, No. 5, September 1980.

[Haas 83] L. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource Based System", *Research Report RJ3765*, IBM Research Laboratory, San Jose, California, January 1983.

[Herman 83] T. Herman and K. M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlock", Department of Computer Sciences, University of Texas, Austin, 78712, February 1983.

[Hoare 78] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, August 1978.

[Holt 72] T. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

[Kumar 85] D. Kumar, "Distributed Simulation", Ph.D. Thesis (in preparation), Department of Computer Sciences, University of Texas, Austin, Texas 78712.

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a

Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[Misra 81] J. Misra and K. M. Chandy, "Proofs of Networks of Processes", *IEEE Transactions on Softaware Engineering*, Vol. SE-7, No. 4, pp. 417-426, July 1981.

[Misra 82a] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 37-43, January 1982.

[Misra 82b] J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, pp. 678-688, October 1982.

[Misra 83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal Canada, August 17-19, 1983.

[Obermarck 80] R. Obermarck, "Deadlock Detection For All Resource Classes", *Research Report RJ2955*, IBM Research Laboratory, San Jose, California, October 1980.

[Obermarck 82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp.187-208, June 1982.

[Owicki 76] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, pp.319-340, 1976.